National Library of Canada
Canadian Theses Service
Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Service des thèses canadiennes

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

NL-339 (r. 88/04) c

Canada

University Of Alberta

Reasoning About Design Tradeoffs And Constraint Satisfaction In Integrated Circuit Design

by

Clayton D. Stafford

A thesis

submitted to the Faculty of Graduate Studies and Research

in partial fulfillment of the requirements for the degree

of Master of Science

Department Of Electrical Engineering

Edmonton, Alberta

Spring, 1991

National Library of Canada | Bibliothèque nationale du Canada

Canadian Theses Service | Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN · 0-315-66583-1

Canada

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Clayton D. Stafford

TITLE OF THESIS:

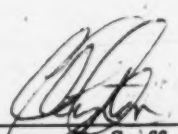
"Reasoning About Design Tradeoffs And Constraint Satisfaction In Integrated Circuit Design"

DEGREE: Master of Science

YEAR THIS DEGREE GRANTED: 1991

Permission is hereby granted to the University of Alberta to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.



Clayton D. Stafford

Box 261
Evansburg, Alberta
Canada
TOE OTO

Date: 91-04-09

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

THE UNDERSIGNED CERTIFY THAT THEY HAVE READ, AND RECOMMEND TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH FOR ACCEPTANCE, A THESIS ENTITLED

"Reasoning About Design Tradeoffs And
Constraint Satisfaction In Integrated Circuit Design"

SUBMITTED BY: Clayton D. Stafford

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE.

Jack Mowchenko
J.T. Mowchenko

Emil Gircyc
E.F. Girczyc

Renée S. Elio
R. Elio

Peter van Beek
P. van Beek

Date:

Abstract

In a design process, a designer attempts to generate a design that satisfies certain constraints; however, the components selected to implement the design sometimes cause these constraints to be violated. In the case of integrated circuit (IC) design, constraints are placed on such properties as the area and signal propagation delay of the completed circuit. A complication to the constraint satisfaction process is the fact that certain types of constraints compete with other types of constraints. As a result, resolving constraint violations often involves reasoning about design tradeoffs between these competing constraints. This is the case with area and delay constraints -- reducing the area of an IC cell will generally increase delay and vice versa.

This thesis presents Viola, a knowledge-based system for resolving area and delay constraint violations in IC designs. When presented with a list of constraint violations, Viola selects the most severe constraint violation from the list and resolves it either by relaxation or by redesigning the IC cell. A domain-independent strategy is applied to decide which course of action to take. The decision-making process is sensitive to the severity of the constraint violation, the presence of any competing constraints and the apparent success or failure of past violation handling activities. This process of resolving one constraint violation at a time is repeated until all constraints are satisfied.

The results of Viola's handling of several constraint satisfaction problems are presented and compared to solutions generated by hand. It is concluded that Viola is a flexible, computationally efficient system for reasoning about design tradeoffs while resolving constraint violations in IC designs. It is also concluded that Viola's clear separation of domain-dependent and domain-independent knowledge and its explicit, domain-independent strategy should make it readily transferrable to other domains.

Acknowledgements

I would like to thank my research advisors, Renee Elio and Emil Girczyc, for their assistance over the course of my research. Both provided invaluable input despite the distance over which we had to communicate. I would also like to thank AGT Limited for the financial support that made this research possible. Finally, I would like to thank my wife, Tammie, for her support and perserverence.

Table Of Contents

Chapter 1. Introduction 1

1.1. Objectives Of Viola's Design 2

1.2. Overview Of The Thesis 4

Chapter 2. Related Work 5

2.1. Systems Without Violation Handling 5

2.2. Systems Using Constraints As Concepts 7

2.3. Other Approaches To Violation Handling 9

2.4. Summary 11

Chapter 3. Constraint Propagation In STEM 13

3.1. Viola And STEM 13

3.2. Constraint Propagation -- The Basics 14

3.3. Area Constraints 15

3.4. Delay Constraints 16

Chapter 4. Viola's Focus Of Attention Stage 18

4.1. Overview Of The Focus Of Attention Stage 20

4.2. Competing Constraint Identification 22

4.3. Constraint Ranking 22

4.4. Bypass Operator Application 24

4.5. Precedence Operator Application 26

Chapter 5. Viola's Plan Proposal Stage 30

5.1. Severity Assessment Operators 32

5.2. Historical Evaluation Operators 33

5.3. Meta-Operators 35

Chapter 6. Viola's Plan Refinement And Implementation Stage 39

6.1. Constraint Relaxation 40

6.2. STEM's Cell Library 40

6.3. Cell Redesign For Area Improvement 43

6.4. Cell Redesign For Delay Improvement 49

Chapter 7. Discussion Of Results 53

7.1. The Subcell Library And The Test Cell 53

7.2. Constraint Satisfaction Problems With A Single Overriding Constraint

7.2.1. Test Case 1 58

7.2.2. Test Case 2 59

7.2.3. Test Case 3 60

7.2.4. Summary Of Results For Type I Problems 61

7.3. Violation Handling With Equally Important Constraints 62

7.3.1. Test Case 4 63

7.3.2. Test Case 5 64

7.3.3. Test Case 6 64

7.3.4. Test Case 7 65

7.3.5. Summary Of Results For Type II Problems 66

Chapter 8. Conclusion 67

8.1. Handling Constraint Violations In IC Design 67

8.2. Discussion 68

8.2.1. Benefits Of Viola's Approach 69

8.2.2. Limitations Of Viola's Approach 73

8.3. Suggestions For Future Research 77

References 80

Appendix I - Test Case Transcripts 82

List Of Tables

Table 4.1. Severity Ratings Used By Viola 23

Table 4.2. Constraints For Example 24

Table 4.3. Precedence Operators 27

Table 5.1. Severity Assessment Operators 32

Table 5.2. Historical Evaluation Operators 34

Table 7.1. Initial Conditions For Type I Problems 56

Table 7.2. Results For Type I Problems 58

Table 7.3. Initial Conditions For Type II Problems 63

Table 7.4. Results For Type II Problems 63

List Of Figures

Figure 3.1. Schematic Representation Of The Design Process 14

Figure 3.2. An Example Delay Network 17

Figure 4.1. Control Flow For Viola 18

Figure 4.2. Control Flow For The Focus Of Attention Stage 20

Figure 4.3. Example Of Severity Information Generated By Viola 24

Figure 5.1. Control Flow For The Plan Proposal Stage 31

Figure 5.2. Control Flow For Meta-Operator Application 37

Figure 6.1. Control Flow For Plan Refinement And Implementation 39

Figure 6.2. Functional Category Inverter 41

Figure 6.3. Functional Category Inverter After Sorting 42

Figure 6.4. Cell Redesign Control Flow For Area Constraints 43

Figure 6.5. A RedesignInfoRecord 44

Figure 6.6. A Violation Handling Example 47

Figure 6.7. Subcell Replacement Scheduling Cycle 48

Figure 6.8. Cell Redesign Control Flow For Delay Constraints 50

Figure 7.1. The Subcell Library 54

Figure 7.2. TestCellA 55

Figure 8.1. An Alternate Subcell Replacement Scheduling Approach 75

Chapter 1. Introduction

Integrated circuit (IC) design may be viewed as a process of top-down refinement where circuit modules at the top level of the design are refined into submodules which, in turn, are refined into smaller submodules. This process continues until the circuit is decomposed into submodules that may be implemented directly using cells from a predefined library. As one circuit module at a given level in the design hierarchy is decomposed and its implementation is completed, the implementation of other modules at that level must be restricted to ensure compatibility with the completed module. In other words, the low-level implementation of a module places constraints on how that module may be used in the circuit. These low-level implementation constraints include such things as minimum area requirements and critical path propagation delays. As in any design process, the circuit designer also places constraints on system performance. In IC design, user-specified constraints would be such things as maximum circuit area and maximum propagation delay.

Constraints specify restrictions on variables associated with circuit modules. As these modules are connected together, a network of interrelated constraints is formed. In order to ensure consistency of this constraint network, it is necessary to propagate variable values from one constraint to another while checking to ensure that the constraints are satisfied. Propagation of variable values occurs within each level of the design hierarchy and also between levels; consequently, it provides a mechanism for verifying that low-level constraints don't conflict with each other or with user-specified constraints at higher levels. This process is called constraint propagation and has been implemented by Ly [10] in the STEM [5] IC design environment at the University of Alberta.

Constraint propagation on its own, however, does not solve the problem of constraint satisfaction. Ly's constraint propagation system identifies constraint violations

in situations where constraint conflicts arise, but doesn't address the problem of resolving the violations through redesign or constraint relaxation -- this is left to the user. An example of a constraint violation is the situation where a user has placed a maximum area constraint on a circuit and the sum of the areas required by the submodules of the circuit exceeds this maximum value. In this situation, the user has two choices -- relax the maximum area constraint or redesign the circuit by replacing one or more circuit submodules. Choosing to relax the constraint will depend on such considerations as the importance of the constraint and whether or not it has been relaxed in the past. Choosing to replace a module requires the designer to consider alternative module implementations available in the cell library and whether or not the available replacements will cause other constraints to be violated.

Reasoning about how to resolve violations in a constraint network is time consuming and requires either expert IC design knowledge or time consuming trial and error. This thesis describes Viola, a constraint violation handler that takes an expert systems approach to reasoning about area and delay tradeoffs while resolving area and delay constraint violations. In the following sections, the primary objectives of Viola's design are presented and the thesis is outlined.

1.1. Objectives Of Viola's Design

The primary objectives in Viola's design were:

(a) The violation handling strategy should be sensitive to constraint violation severity and the presence of competing constraints.

(b) The strategy should be sensitive to the violation handling history of the design.

(c) The strategy should be as flexible and explicit as possible.

(d) There should be a clear separation of domain-dependent and domain-independent knowledge to allow transfer to other domains.

(e) The system should be computationally efficient to be useful for large designs.

(f) The system should enable an inexperienced designer to produce better designs in a shorter time and should reduce the time required for an experienced designer to produce good designs.

Objectives (a) and (b) provide Viola with its ability to reason about the best way to resolve a constraint violation -- relax the constraint or redesign the cell. Viola must not arbitrarily carry out the same corrective action every time a given constraint is violated -- it must consider the severity of the constraint violation, the presence of competing constraints (i.e. constraints that may be adversely affected by redesigning the cell to resolve the violated constraint) and the historical significance of its actions. The more severe a constraint violation is, the less acceptable it will be to simply relax the violated constraint and the more likely it will be that cell redesign is the best course of action. Conversely, the presence of a competing constraint will tend to bias the decision in favour of relaxation to avoid adversely affecting that competing constraint. In addition to this type of reasoning about the severity of a constraint violation versus the presence and severity of competing constraints, Viola must also be able to look at what it has done in the past. In this way, it can avoid repeating past mistakes, repeat actions that have worked before and avoid unnecessarily undoing previous corrective actions.

The reason behind objective (c) is that a flexible and explicit strategy is more readily understood and modified by others. It is necessary for designers to be able to add new knowledge and to remove knowledge that no longer seems appropriate in order to keep the system current. It is important that in doing so the designer is able to understand how the new knowledge will interact with the existing knowledge and how the removal of knowledge will affect the overall performance of the system.

Although Viola was designed to resolve constraint violations in the domain of IC

design, it was considered important that Viola be transferrable to other domains. To facilitate this, objective (d) states that there must be a clear separation between domain-dependent and domain-independent knowledge. The domain-independent knowledge can then be transferred directly to the new domain -- only the domain-dependent knowledge must be modified. For this reason, it is also desirable that the system use as little domain-dependent knowledge as possible.

The final objectives, (e) and (f), are obvious requirements of a good design automation tool.

1.2. Overview Of The Thesis

This thesis describes how Viola was implemented and evaluates it both on the basis of its performance on a number of test cases and on how well it met the objectives laid out in the previous section. The organization of the thesis is as follows. Chapter 2 looks at previous work done on constraint violation handling and Chapter 3 provides some background information necessary to appreciate Viola's approach. A detailed description of Viola's implementation is presented in Chapters 4 through 6. Chapter 7 discusses the results obtained by Viola on a number of sample constraint satisfaction problems. Finally, Chapter 8 concludes with a discussion of the strong and weak points of Viola's approach to constraint violation handling and makes some suggestions for future research.

Chapter 2. Related Work

A number of systems have been developed in recent years that use constraints to guide a variety of design, planning and synthesis tasks. The need to handle constraint violations or failures as they arise has been met to varying degrees of success. The systems discussed in section 2.1 do not handle constraint violations. Section 2.2 examines four systems that have represented constraints as concepts with violation handling knowledge in the representation of the constraints themselves. The systems in section 2.3 have represented knowlege about violation handling external to the constraints. This last approach is closer to the approach taken by Viola.

2.1 Systems Without Violation Handling

As mentioned in Chapter 1, the constraint propagation mechanism developed by Ly [10] for the STEM [5] environment is a useful tool but is incomplete without some means of resolving constraint violations as they arise. A number of other systems have been developed that suffer from the same lack of violation handling capability.

CONSTRAINTS was developed by Sussman and Steele [19] as a language for expressing constraints as algebraic relations between variables. It propagates constraints by performing algebraic manipulations to solve the constraint equations. This system, however, does not address the situation where it is impossible to satisfy all constraints. Rather, it operates under the assumption of a satisfiable constraint network and suffers from the same lack of violation handling mechanism as Ly's system.

Steinberg's VEXED system [11,16] is an interactive digital circuit design aid that allows a user to build circuits using top-down refinement of circuit modules with constraint propagation for consistency maintenance. VEXED's CRITTER subsystem [9] actually performs the constraint propagation. Constraints are derived from the

functional specification of the modules in question and CRITTER checks for constraint violations by simulating the circuit's behaviour and comparing this simulation with the functional specification. Rather than allow the user to refine the circuit in a way that would create constraint violations, the constraints identified by CRITTER are considered absolute and VEXED restricts the user to making modifications that do not violate constraints. This eliminates the view of constraints as competing and potentially unsatisfiable goals and constraint violation handling as a process of reasoning about design tradeoffs in trying to meet these goals.

MOLGEN [17,18] is an expert system that plans gene cloning experiments in molecular genetics. It plans hierarchically by proposing abstract experimental operators and refining them similar to the way that an IC may be designed by top-down refinement. As MOLGEN refines operators, constraints are placed on the inputs to those operators and are propagated up through the plan hierarchy. MOLGEN is a well-engineered system, but it too lacks any violation handling mechanism -- if conflicting or insoluble constraints are discovered, control is returned to the user.

All of these systems use constraint propagation as a mechanism for managing design interactions; however, the system designers chose not to address the problem of constraint violation handling. As Viola was designed as an add-on violation handler for STEM, similar violation handling systems could be added to both CONSTRAINTS and MOLGEN with few modifications to their current frameworks. The VEXED/CRITTER system, on the other hand, would require more extensive modification to incorporate a violation handling capability. Because it uses constraints to limit the user's actions rather than letting the user violate constraints, VEXED is never placed in a position of having to resolve violated constraints.

2.2 Systems Using Constraints As Concepts

One approach to violation handling is to place violation handling knowledge within the constraints themselves. The common problem faced by all systems using this approach is that all problem situations must be anticipated -- there is no reasoning about the best way to resolve a constraint violation based on the problem state at the time the constraint is violated.

In ThingLab, Borning [1] uses this approach. ThingLab is a simulation system based on constraints and constraint propagation. A user defines objects and constraints on those objects and constraint propagation verifies the correctness of a design as the user connects objects together. Constraints in ThingLab consist of a rule that must be satisfied and a set of methods for satisfying the constraint. The order of the methods indicates a preference. When presented with a design change, ThingLab uses a constraint satisfier to develop a plan for satisfying any constraints affected by the change. The plan consists of calls to the various predefined constraint satisfaction methods. If a plan cannot be developed to satisfy all constraints, ThingLab relaxes all unsatisfied constraints by approximating them as linear equations and finding a least-mean-squares fit. This often results in more constraints being relaxed than would be necessary if there were some mechanism for reasoning about the most appropriate relaxations to make.

Fox [4] recognizes that constraints may not always be satisfied and identifies some additional information which must be taken into consideration in violation handling: how important is the constraint; can the constraint be relaxed; how should it be relaxed; how will relaxation affect other constraints; when shouldn't it be relaxed.

In his ISIS system, Fox encodes this information in a symbolic representation of constraints as concepts. The way in which a constraint can be relaxed is prespecified -- all

of the reasoning about how to relax a constraint is done ahead of time; therefore, the system is incapable of adjusting to the success or failure of its previous constraint satisfaction actions. ISIS also uses some rules to encode knowledge about violation handling, but the rules too are insensitive to design history.

Brown and Chandrasekaran [2] address the need for explicit knowledge about violation handling, which they call failure recovery, in the AIR-CYL system. AIR-CYL is an expert system for the mechanical design of air cylinder systems. A design is executed by a number of "specialists" that check various constraints and call on other specialists. If one of these constraints is violated, the specialist receives a message from the constraint containing suggestions about what to do to fix the problem. As in Fox's approach, these suggestions are predefined with the constraints themselves. The message is passed to a failure handler associated with the specialist that may attempt redesign using the suggestions. This is a hard-coded, inflexible violation handling scheme that fails to consider constraint relaxation as an alternative. Brown and Chandrasekaran acknowledge the need for some means of reasoning about how to resolve constraint violations as they arise.

PRIDE [12] is an expert system for the design of paper handling systems that uses constraints to verify that a solution or partial solution is acceptable. Each constraint contains information about whether or not it may be relaxed and also advice about what to do if it is violated. This system improves on AIR-CYL by considering constraint relaxation as an alternative. The designers of the PRIDE system also state that some means of reasoning automatically about which constraints to relax and how is needed.

All of these systems use predefined, domain-specific knowledge encoded in the constraint representations to handle constraint violations when they arise. As a result, they are unable to react to differences between problem situations and reason about the

best way to resolve them. Also, none of the systems consider design as a process with a history of decisions; therefore, they lack a general strategy.

2.3 Other Approaches To Violation Handling

Unlike the above systems, several other approaches have represented violation handling knowledge separate from the constraints. For example, Gray [8] has created a design system called Diadem that uses constraint propagation and addresses the problem of resolving constraint violations. A truth maintenance system (TMS) is used to record all design decisions made. Constraints represented as rules which, if fired, trigger a "contradiction" condition. When a contradiction arises, the design stops and dependency-directed backtracking is initiated. The TMS is inspected to find a design decision that may be retracted in order to satisfy the constraint. Diadem considers constraints to be absolute restrictions and fails to consider that, in some cases, it may be better to retract a different design decision or to simply relax the violated constraint.

PROMPT [13] is a system that uses constraint propagation to design physical systems. An example application is beam design. The problem of constraint satisfaction is handled with two approaches:

1) PROMPT first tries to vary parameters of the object currently being used. An example of this is changing the diameter of a solid circular beam to increase its strength.

2) If the first approach fails, PROMPT uses "heuristic modification operators" to redesign the object. These operators represent knowledge about how to resolve specific problems. An example is a mass redistribution operator that may decide to resolve a conflict between mass and strength by replacing a solid circular beam

with a hollow pipe to increase strength without increasing mass.

Both of these approaches use mathematical equations to determine the characteristics of the redesigned object. The problem with applying this type of approach to IC design and to many other design tasks is that there are not usually mathematical equations that can be used to modify objects and vary parameters. For example, there are many ways to reduce the area of an IC cell, each of which may have a different impact on the cell's propagation delay. For the most part, modifying an IC will involve replacing one module with another module of similar function -- a discrete change whose effects can be computed after the fact but cannot be represented mathematically. The heuristic operators also hide any overall violation handling strategy and make any variation of that strategy difficult.

TLTS [15] diagnoses signal integrity problems of completed printed circuit board designs and redesigns them to eliminate the problems. It simulates the circuit's behaviour and compares it with user constraints, process constraints, etc. to look for problems. If a constraint is violated, a knowledge source associated with that type of violation will try to redesign the circuit. This knowledge source contains redesign operators that it uses to modify the circuit. The obvious problem with this system is that a knowledge source must be set up for every possible type of constraint violation and redesign operators must be set up for every possible way of redesigning the circuit to eliminate these violations.

The MICON system [3] designs small computer systems by top-down refinement of a high-level functional specification. As the design proceeds, constraints are used to maintain consistency between subsystems and ensure that top level specifications are met. Constraint violations occur when there are no parts available that will satisfy the specifications. In the event of a violation, the system performs dependency-directed backtracking to determine the source of the failure. The user is then asked how to

correct the failure. As the user tells MICON how to solve the problem, a rule is added to the database so that any time this constraint is violated in the future, it will be handled in exactly the same way. MICON, then, is not capable of handling any problem that it hasn't seen before without user intervention. It is simply capable of repeating previous solutions.

These systems take an approach similar to that taken by the systems in Section 2.2 but the violation handling knowledge is contained outside of the constraints. The major problem with these systems is, once again, the inability to reason about the best way to resolve a constraint violation when it arises. Users of all of these systems must anticipate problems before they occur and provide the system with knowledge about how to resolve them.

2.4 Summary

The fundamental problem with previous attempts at constraint violation handling is their lack of flexibility. None of the systems discussed in this chapter reason about the best way to resolve a constraint violation -- they simply respond to problem situations by performing a predefined corrective action. Because the problem-solving in these systems is done in advance, they are insensitive to differences between one problem situation and another such as differing degrees of violation, the existence of other constraints and the effect of any prior constraint violations on the design. As a result, they will always respond to the violation of a given constraint in the same way regardless of differences in other aspects of the design.

Viola, on the other hand, makes use of general problem-solving knowledge to look at the violated constraint, its current environment and the design's history of constraint violations. The result is a more flexible system that is capable of reacting to

subtle differences between constraint violations.

Chapter 3. Constraint Propagation In STEM

This chapter describes how constraint propagation is handled in STEM, the environment in which Viola was built. Since Viola was designed to be integrated into the STEM environment, some design decisions were influenced by STEM's operation. For that reason, it is important to understand what STEM is, how it works and what role Viola was intended to fulfill. Section 3.1 talks about where Viola fits in with STEM and how they interact. Section 3.2 describes variables and constraints -- the participants in the constraint propagation process. Finally, sections 3.3 and 3.4 look at the area and delay constraints that Viola was designed to reason about.

3.1 Viola and STEM

STEM, the SmallTalk Environment for Module design, [5] is an integrated circuit design environment implemented in the Smalltalk-80 object-oriented programming language [6,7]. It provides both a common data structure for representing IC cells and a mechanism for maintaining consistency between different views opened on this data structure by IC design tools. In this way a number of different tools can be integrated into a common environment. The IC design tool relevant to Viola is one that allows a designer to specify the structure of a cell interactively.

To construct a cell, the designer selects subcells from an existing cell library, places them, and connects them together. As structure is added to the cell, a constraint propagation system within STEM incrementally updates all system variables and checks any constraints on the cell. The designer may also add single transistors of various dimensions to the cell. This design process is represented schematically in Figure 3.1.

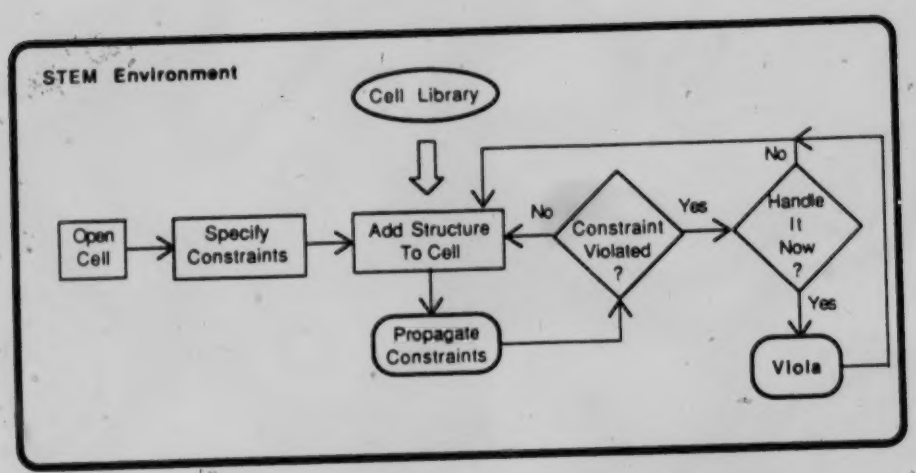


Figure 3.1. Schematic Representation Of The Design Process

When a constraint is violated, the designer is asked whether he wants to continue with the design or handle the constraint violation now. If he chooses to handle the constraint violation, the Viola subsystem is called upon. If not, he continues with the design and may initiate violation handling manually at any time. Viola will then resolve any constraint violations on the cell either by relaxing the violated constraints or by redesigning the cell.

3.2. Constraint Propagation -- The Basics

Constraint propagation is a process of propagating data from one point in a network to other related points in the network. The points in the network are variables, the links between points are constraints and the network itself is called a constraint network.

In STEM, variables are objects used to store data about integrated circuit cells. This work is concerned only with delay and area variables. A constraint specifies restrictions on the values of one or more variables. The variables associated with a constraint are called its arguments.

When the value of a variable is changed, its new value is propagated to all associated constraints. When a constraint receives a propagation message, it assigns new values to any arguments affected by the changed variable. These variables, in turn, propagate their new values to any other associated constraints.

This process continues until the initial change has been completely propagated. At that time, each constraint that the propagation affected is tested for satisfaction. If a constraint fails this test, a constraint violation has occurred. It is this type of constraint violation that Viola has been designed to resolve.

3.3. Area Constraints

The area of a cell is derived from the areas of its subcells using the following relation:

$$CellArea = [\sum(SubcellAreas)] x (1+ConnectionFraction)$$

where the ConnectionFraction is an approximation of the fraction of the cell area that would be taken up by connections between subcells if the layout were compacted as much as possible. Area is calculated in this way rather than using the actual area of the user's layout because the layout must be quite sparse to allow room for potential subcell replacements by Viola. The value of ConnectionFraction may be assigned by the user (a value of 0.30 was used for this thesis).

If a cell is a primitive (i.e. a single transistor), it has no subcells and its area is just calculated directly as the area of a bounding box around the primitive.

The user can constrain the area of a cell by specifying a constraint called a *MaximumAreaPredicate*. This constraint places an upper limit on the cell's area.

3.4. Delay Constraints

The simple delay model used in STEM operates on the assumption that the delays of consecutive elements in a delay path are additive. The delay values are also adjusted to account for the output resistance of output signals and the loading capacitance of input signals along the delay path. The delay of a signal along a path, then, is a sum of subcell delay values plus an additional RC delay value for each connection between subcells along the path.

If more accurate delay values are desired for the lower level cells, they may be determined through simulation with SPICE [20] or an equivalent system and may be specified as default delay values. Because these lower level cells are to be used many times in designing larger cells, more accurate delay values are desirable.

Subcell delays are combined to obtain overall cell delays using two types of constraints -- *UniMaximumConstraints and UniAdditionConstraints*. An example of a delay network for a simple cell is presented in Figure 3.1. Variables are represented as ovals, constraints are represented as rectangles and arrows indicate interactions between variables and constraints.

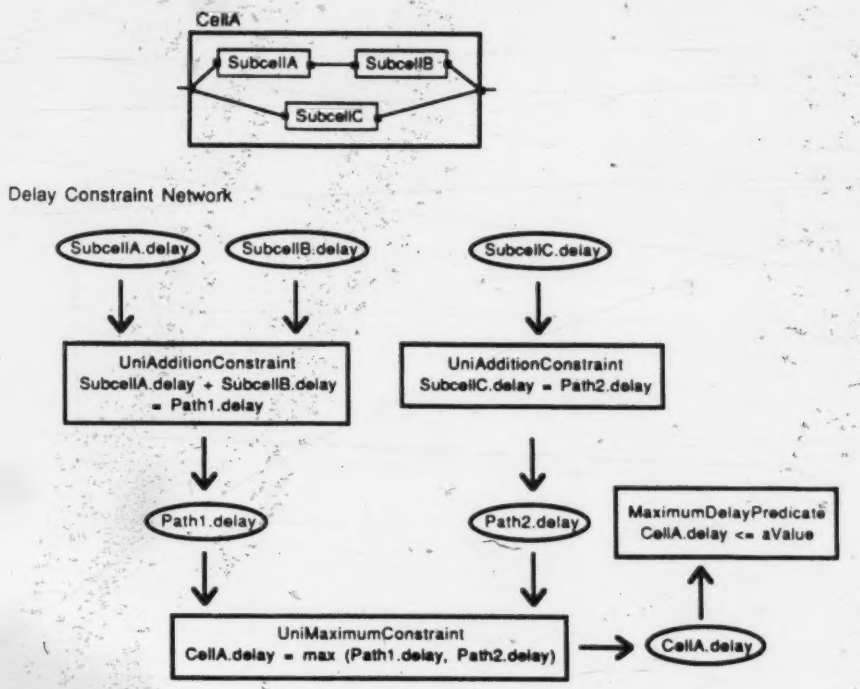


Figure 3.2. An Example Delay Network

For CellA, there are two delay paths. The overall cell delay is the delay value of the longer of these two paths. The *UniAdditionConstraints* are used to add up the subcell delays (including RC adjustment) along a single delay path. Once the path delays have been calculated, a *UniMaximumConstraint* selects the maximum path delay and assigns it to the cell's delay variable. The user may then constrain this delay variable using a *MaximumDelayPredicate* to place an upper limit on the delay value. For any of the cases studied in this research, only the *MaximumAreaPredicates* and the *MaximumDelayPredicates* were violated.

The remainder of this thesis looks at the approach taken by Viola to reasoning about the interactions between area and delay in resolving constraint violations on IC cells.

Chapter 4. Viola's Focus Of Attention Stage

Viola's task is to take, as input, a list of one or more constraint violations and resolve them. Viola operates in a serial fashion, resolving one constraint at a time until all constraints are satisfied as shown in Figure 4.1.

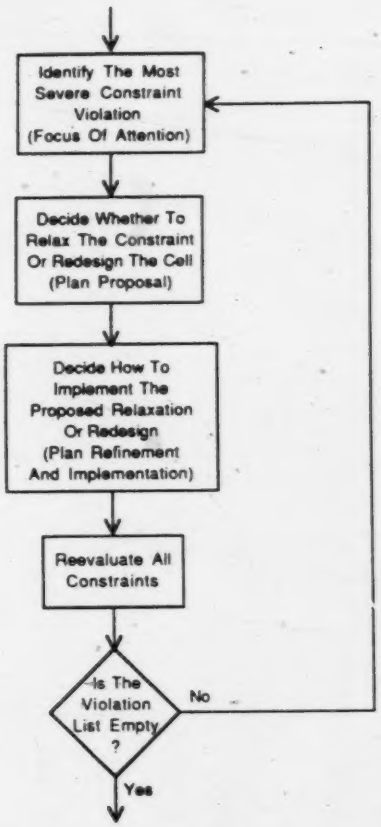

Identify The Most Severe Constraint Violation (Focus Of Attention)
Decide Whether To Relax The Constraint Or Redesign The Cell (Plan Proposal)
Decide How To Implement The Proposed Relaxation Or Redesign (Plan Refinement And Implementation)
Reevaluate All Constraints
Is The Violation List Empty ?
No
Yes

Figure 4.1. Control Flow For Viola

To resolve a single contraint violation, Viola works through three stages. First, Viola decides which of possibly several violated constraints it will first work on. This is called the *focus of attention* stage. This decision is based on an assessment of the relative severity of the constraint violations; thus, the purpose of the focus of attention stage is to select the most severe constraint violation.

The second stage of Viola's operation is the *plan proposal* stage. In this stage, Viola chooses between two courses of action that may be taken to resolve the selected constraint violation -- relaxing the violated constraint and redesigning the cell.

The final stage of Viola's operation is *plan refinement and implementation*. In plan refinement and implementation, Viola determines how to relax the constraint or redesign the cell depending on which of these alternatives it chose in the preceding stage. If the plan is to relax the constraint, this stage is trivial because the constraint can simply be relaxed just enough so that it is satisfied by the current value of its arguments. If, on the other hand, the plan is to redesign the cell, a more detailed plan must be laid out to determine exactly which subcells to replace in redesigning the cell.

These three stages represent Viola's effort to resolve a single constraint violation -- upon completion, it has either relaxed the constraint or redesigned the cell. Viola's actions are likely to have impacted other constraints -- both those on the original violation list and others that were not violated before this process was begun. At this point, the constraint propagation system determines which constraints, if any, are now violated and a new violation list is constructed. If the violation list contains no constraint violations after testing all of the constraints, violation handling ends and control is returned to the user.

This chapter and the following two chapters describe the three stages of Viola's operation. The focus of attention stage is described in this chapter, the plan proposal stage is described in Chapter 6 and the plan refinement and implementation stage is described in Chapter 7.

4.1. Overview Of The Focus Of Attention Stage

Figure 4.2 shows the control flow for Viola's focus of attention stage. To determine which constraint violation is the most severe, Viola considers not only the constraints on the violation list but also the environment in which those constraints exist. A constraint's environment is characterized by its interaction with other constraints. In particular, Viola is designed to identify constraints that *compete* with one another. One constraint is said to compete with another if improving the value of its argument has a potential detrimental effect on the other constraint's argument. In integrated circuit design, area and delay are competing quantities -- reducing the area of a cell will generally increase the propagation delay of signals passing through that cell and vice versa.

Identify Competing Constraints For All Constraints On The Violation List
Rank All Violated Constraints And Their Competitors
Apply Bypass Operators And Flag Trivial Violations
Use Precedence Operators To Identify The Most Severe Constraint Violation

Figure 4.2. Control Flow For The Focus Of Attention Stage

At first glance, it seems that selecting the most severe constraint violation is simple; however, the selection is complicated by the presence of competing constraints. Fixing a constraint violation may cause a competing constraint to be violated. So the process of selecting the most severe constraint violation is really a delicate balance between selecting the most important constraint and selecting the constraint whose solution will cause the least detriment to other constraints.

To enable selection of the most severe constraint violation, Viola first identifies all of the constraints that compete with each constraint on the violation list. It then ranks each of the violated constraints and each of their competing constraints on the basis of their user-specified importance ratings and their violation margins. The importance rating indicates the relative importance of each constraint and the violation margin is a measure of how close the constraint is to being violated.

Once all of the violated constraints and their associated competitors are ranked, Viola proceeds with the selection of the most severe constraint violation. To avoid wasting time considering constraint violations that are obviously unimportant (i.e. constraints that have a low importance rating and are violated by a small amount), Viola uses a group of *bypass operators* to screen out these trivial violations before proceeding with the decision-making process. The system will fix a trivial constraint violation only if there are no non-trivial constraint violations on the violation list.

The final selection of the most severe constraint violation is made by a group of *precedence operators* that represent knowledge about what conditions are sufficient to declare one constraint violation more severe than another.

The next four sections describe the steps involved in focussing Viola's attention on the most severe constraint violation -- competing constraint identification, constraint ranking, bypass operator application and precedence operator application.

4.2. Competing Constraint Identification

In order for Viola to identify the competitive interactions between constraints on a cell, it requires knowledge about which types of constraints compete. This knowledge is represented in the constraints themselves. The competitive relationships are defined in terms of constraint types -- an area constraint knows that it competes with any delay constraints on the same cell and a delay constraint knows that it competes with any area constraints on the same cell. These competitive relationships are inherited by all of the specific area and delay constraints on a cell.

As Viola proceeds with the focus of attention stage, the competing constraints of each constraint on the violation list are identified and stored. This provides Viola with some of the information that it needs to identify the most severe constraint violation; however, it still needs information about the individual importance of each constraint on the violation list and the importance of their competing constraints.

4.3. Constraint Ranking

The severity of a constraint violation is based on two primary factors -- the importance of the violated constraint and the extent of its violation. A third consideration is the importance and possible violation of its competitors.

The importance of a constraint is specified by the user when the constraint is placed on the cell and may have one of three values: important (I), very important (VI) or not important (NI). The extent of violation is called the violation margin and is expressed as a percentage of the constraining value. Expressing the violation margin as a percentage allows a meaningful comparison of area and delay constraints despite the fact that area and delay are expressed in different units.

The severity ratings used by Viola are presented in Table 4.1. In this table, a higher numerical severity rating indicates a greater severity. For a constraint to be assigned a severity rating, it must have a certain importance value and its violation margin must fall within a certain range. The range of accepted violation margins is delimited by a lower violation margin and an upper violation margin. In Table 4.1, a negative violation margin indicates an unviolated constraint. If a severity rating has a lower violation margin of nil, this indicates that the range of violation margins for that severity rating has no lower bound. Similarly, if a severity rating has an upper violation margin of nil, there is no upper bound.

As the severity ratings are listed in Table 4.1, the importance rating of a constraint is considered more significant than its extent of violation. For example, a very important (VI) constraint violated by 10% would receive a severity rating of 11 and an important (I) constraint violated by 30% would receive a severity rating of only 10. However, this is a design decision that can be changed -- the severity ratings are readily modified by the user.

Table 4.1. Severity Ratings Used By Viola

Severity Rating	Importance	Lower Violation Margin	Upper Violation Margin
12	VI	25.0%	nil
11	VI	0.0%	25.0%
10	I	25.0%	nil
9	I	0.0%	25.0%
8	NI	25.0%	nil
7	NI	0.0%	25.0%
6	VI	-25.0%	0.0%
5	VI	nil	-25.0%
4	I	-25.0%	0.0%
3	I	nil	-25.0%
2	NI	-25.0%	0.0%
1	NI	nil	-25.0%

As a further example, consider the three constraints listed in Table 4.2. Initially,

constraints C1 and C3 are on the violation list. After identifying the competing constraints of C1 and C3 and ranking all of the constraints, the information gathered by Viola is as shown in Figure 4.3. Sufficient information is now available about the severity of the individual constraints and their interaction with other constraints for Viola to proceed with its selection of the most severe constraint violation from the violation list. Before making this selection, however, it is desirable to remove any obviously trivial constraint violations from consideration. This is the role of the bypass operators.

Table 4.2. Constraints For Example

Name	Type	Importance	Violation Margin
C1	area	VI	+3.0%
C2	delay	VI	-2.0%
C3	delay	I	+7.0%

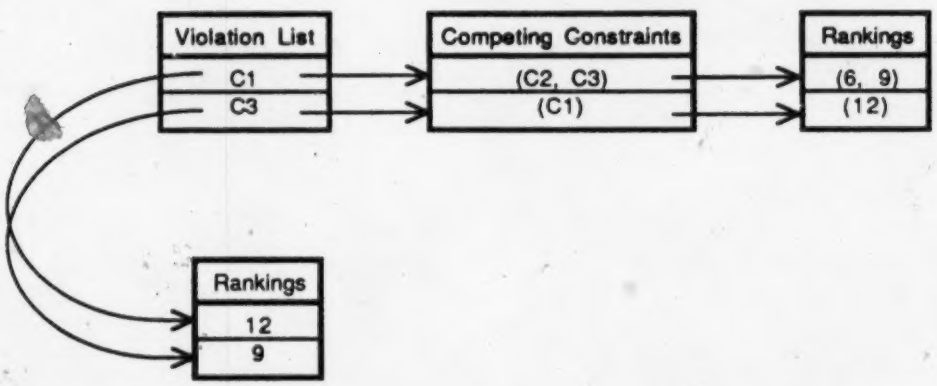


Figure 4.3. Example Of Severity Information Generated By Viola

4.4. Bypass Operator Application

At a high level, a circuit designer faced with a constraint violation makes a decision about whether or not the violation is severe enough to justify going through an

exhaustive decision-making process to decide how to resolve it. If a constraint violation is insignificant or trivial, the designer will just relax it and proceed with his design. If there are other, non-trivial constraint violations, the designer will proceed with resolving them and ignore the trivial constraint violation.

Viola uses a pair of bypass operators to identify trivial constraint violations before proceeding with the decision-making process. Each bypass operator encodes a piece of knowledge about when a constraint violation should be considered trivial. The two pieces of knowledge encoded in Viola's bypass operators are:

BypassOperator1

If:
    A violated constraint has no competing constraints
 & Its ranking is less than a threshold ranking
Then:
    it is trivial.

BypassOperator2

If:
    A violated constraint has one or more competitors
 & Its ranking is less than the ranking of its highest ranked competitor
    by greater than a threshold value
Then:
    it is trivial.

BypassOperator1 basically screens out any constraint violations below a certain severity rating and BypassOperator2 avoids working on a constraint violation if resolving it could potentially damage a much more important constraint. A design feature that adds flexibility to Viola is the ability of the user to activate or deactivate each bypass operator as required. Only the activated bypass operators are used by Viola to identify trivial constraint violations.

At this stage, any constraint violations identified as trivial will be flagged as such for future reference. Flagging a constraint violation as trivial effectively bypasses any

decision-making as far as that constraint is concerned for the current violation handling pass. Precedence operator application will ignore trivial constraint violations -- only if there are no non-trivial violations will a trivial violation be selected by Viola as the most severe constraint violation. If Viola does decide to resolve a trivial constraint violation, the plan proposal stage will be bypassed and the constraint will be relaxed directly. This will be discussed further in the next chapter.

At this point, the competing constraints have been identified, all constraints have been ranked and any trivial violations have been identified. Viola may now proceed with the selection of the most severe constraint violation.

4.5. Precedence Operator Application

To identify the most severe constraint violation from a group of constraint violations, a circuit designer uses knowledge about conditions that are sufficient to declare one constraint violation more severe than another. For Viola, this type of knowledge is encoded in the precedence operators.

The fundamental premise underlying all of the precedence operators is that the more severe a violated constraint's competing constraints, the less severe the constraint becomes by comparison. This recognizes that any activity taken to improve a constrained quantity will have potential adverse effects on any competing constraints. The more severe these competing constraint are, the less acceptable it is to adversely effect them and the less important it becomes to resolve the original constraint violation. A secondary premise behind the precedence operators is that the severity of a constraint's highest ranked competitor is more important than the number of competing constraints that it has -- it would be more damaging to adversely affect a single very important constraint than it would be to adversely affect several less important constraints. The knowledge represented by Viola's precedence operators is stated in Table 4.3.

Table 4.3. Precedence Operators

Precedence Operator	Conditions (VC = violated constraint, CC = competing constraint) VC A is more severe than VC B if: ...
PrecedenceOperator1	- neither VC has CC's - VC A is higher ranked than VC B
PrecedenceOperator2	- only VC B has CC's - VC A and VC B are equally ranked
PrecedenceOperator3	- both VC's have CC's - VC A and VC B are equally ranked - VC A has a lower ranked top-ranking competitor
PrecedenceOperator4	- both VC's have CC's - VC A and VC B are equally ranked - the top-ranking CC's of the two VC's are equally ranked - VC A has a smaller sum of CC rankings
PrecedenceOperator5	- both VC's have CC's - the top-ranking CC's of the two VC's are equally ranked - VC A is higher ranked than VC B
PrecedenceOperator6	- both VC's have CC's - VC A is higher ranked than VC B - VC A has a lower ranked top-ranking CC
PrecedenceOperator7	- both VC's have CC's - VC A is higher ranked than VC B - VC A's top-ranking CC is higher ranked than VC B's top-ranking CC - the difference between VC rankings is greater than the difference between the rankings of the top-ranking CC's
PrecedenceOperator8	- both VC's have CC's - VC A is lower ranked than VC B - VC B's top-ranking CC is higher ranked than VC A's top-ranking CC - the difference between VC rankings is less than the difference between the rankings of the top-ranking CC's
PrecedenceOperator9	- both VC's have CC's - the higher ranked VC has the higher ranked top-ranking CC - the difference between VC rankings is equal to the difference between the rankings of the top-ranking CC's - VC A has the smaller sum of CC rankings
PrecedenceOperator10	- both VC's have CC's - VC A and VC B are equally ranked - the top-ranking CC's of the two VC's are equally ranked - the sum of CC rankings for the two VC's are equal - VC A has a greater violation margin

The precedence operators in Table 4.3 are listed in order of increasing complexity. PrecedenceOperator1 checks for the simplest condition where neither of the constraints has competitors, PrecedenceOperator2 checks for the condition where only one of the constraints has competitors and the remaining operators are concerned with conditions where both constraints have competitors.

Precedence operators 3, 4 and 10 are concerned with the situation where the two constraints are equally ranked. PrecedenceOperator3 tries to differentiate on the basis of the rankings of the top-ranking competitors. If this fails, PrecedenceOperator4 tries to differentiate by considering all of the competitors of the two constraints. Finally, PrecedenceOperator10 tries to make a decision based on the actual violation margins of the constraints.

Precedence operators 5 to 9 are concerned with the situation where the two constraints are not equally ranked. If the top-ranking competitors are equally ranked, PrecedenceOperator5 can select the most severe constraint. If, however, the top-ranking competitors are not equally ranked, the decision becomes more difficult. Since a high constraint ranking and a low competitor ranking both favour the same choice, PrecedenceOperator6 can differentiate between constraints in the situation where the higher ranked constraint has the lower ranked top-ranking competitor. If the higher ranked constraint also has the higher ranked top-ranking competitor, the comparison shifts to the difference between constraint rankings and the difference between top-ranking competitor rankings. Precedence operators 7 and 8 try to differentiate on that basis. If all else fails, PrecedenceOperator9 tries to differentiate by bringing all of the competitors of both constraints into consideration.

As with the bypass operators, the user may activate or deactivate the precedence operators as he sees fit. By stepping through the violation list, applying the active precedence operators to pairs of constraints and keeping track of the most severe

constraint violation as it goes, Viola is able to identify the most severe constraint violation. Once this has been completed, the focus of attention stage is complete. Viola then takes the most severe constraint violation and proceeds to the plan proposal stage where a decision will be made about whether to relax the selected constraint or redesign the cell in order to resolve the constraint violation.

Chapter 5. Viola's Plan Proposal Stage

In the plan proposal stage, Viola decides either to relax the violated constraint identified in the focus of attention stage or to redesign the cell. The first alternative, constraint relaxation, loosens the violated constraint until it is satisfied for the current cell design. The second alternative, cell redesign, leaves the violated constraint as it is and attempts to satisfy it by redesigning the cell.

The decision of how to resolve a constraint violation is sensitive to (a) the severity of the constraint violation and (b) the history of violation handling activity for the cell. Viola uses two sets of operators to "vote" on the course of action it should take -- *severity assessment operators* (SAO's) and *historical evaluation operators* (HEO's). The SAO's make recommendations based on the severity of the constraint violation and its competing constraints and the HEO's make recommendations based on the actions taken by Viola in the past. Each SAO and HEO determines whether it has any relevant input for the current decision. If it does, it casts a vote for either relaxation or redesign depending on which course of action it favours.

The control flow for the plan proposal stage is presented in Figure 5.1. As stated in the last chapter, the plan proposal stage is bypassed for trivial constraint violations. Once the SAO's and HEO's have been applied and a number of votes have been cast for relaxation and redesign, it would seem that Viola could directly make a decision by counting the votes in favour of the two alternatives. However, some of the voters (the SAO's and HEO's) are more important than others. The knowledge of which voters are more important is represented in a group of *meta-operators*. Once these meta-operators have been used to weed out less important voters, Viola can count up the remaining votes and make its decision.

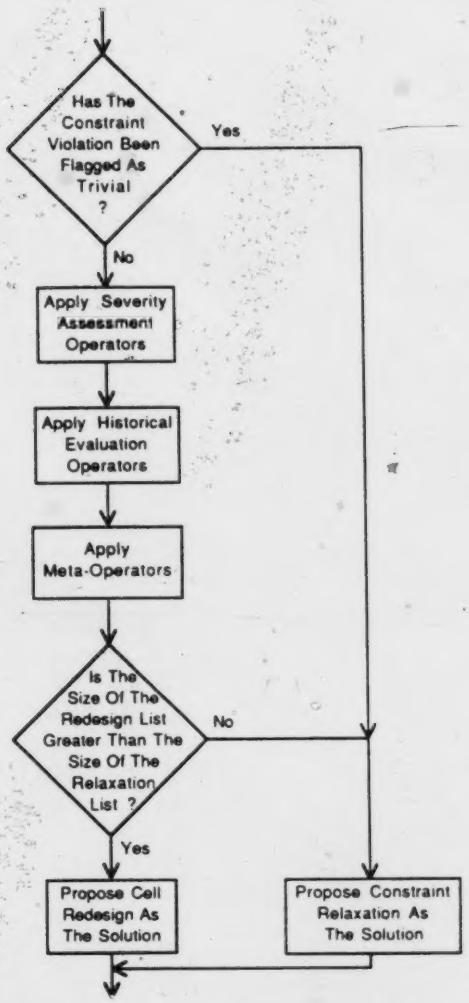

Has The Constraint Violation Been Flagged As Trivial ?
Yes
No
Apply Severity Assessment Operators
Apply Historical Evaluation Operators
Apply Meta-Operators
Is The Size Of The Redesign List Greater Than The Size Of The Relaxation List ?
No
Yes
Propose Cell Redesign As The Solution
Propose Constraint Relaxation As The Solution

Figure 5.1. Control Flow For The Plan Proposal Stage

The following sections look at each of the operator types used to make the relax/redesign decision -- severity assessment operators, historical evaluation operators and meta-operators.

5.1. Severity Assessment Operators

When deciding whether to relax or redesign, a number of factors related to constraint violation severity come into play. In Viola, this knowledge about violation severity is encoded in the SAO's. Each SAO favours either relaxation or redesign. In general, the knowledge in the SAO's is based on the premise that the greater the extent of the violation, the greater the likelihood that redesign is the appropriate decision. Similarly, the more important a constraint's competitors, the greater the likelihood that relaxation is the appropriate decision. The knowledge contained in Viola's SAO's is listed in Table 5.1.

Table 5.1. Severity Assessment Operators

SAO	If: (Conditions) Then Favour: (Recommendation) (VC = violated constraint, CC = competing constraint)	
	Conditions	Recommendation
SAO1	- the current VC ranking is greater than a threshold value	redesign
SAO2	- the current VC ranking is less than a threshold value	relaxation
SAO3	- the VC has one or more CC's - the ranking of the VC is greater than the ranking of its top-ranking CC	redesign
SAO4	- the VC has one or more CC's - the ranking of the VC is lower than the ranking of its top-ranking CC	relaxation
SAO5	- the VC has more than one CC - the sum of CC rankings is greater than the VC ranking	relaxation

SAO's 1 and 2 make recommendations based solely on the ranking of the violated constraint. SAO's 3 and 4, on the other hand, look at both the violated constraint's ranking and the ranking of the top-ranking competitor; therefore, SAO's 3 and 4 should be considered more important. This distinction is made by the meta-operators (to be discussed later). Since Viola was designed to consider the top-ranking

competitor more important than any other competitors, meta-operators will also identify SAO5 as less important than SAO's 3 and 4.

An SAO's applicability is defined by its conditions. Any SAO whose conditions are satisfied will register a vote. The votes are registered by adding the voting SAO to either a *redesign list* or a *relaxation list*. Once any applicable SAO's have cast their votes, Viola's next step is to get input from the HEO's.

5.2. Historical Evaluation Operators

As a designer handles a constraint violation on a cell, he considers what he has done to resolve previous constraint violations on that cell before he makes a decision about how to resolve the current violation. It may be that the current constraint has been violated before. If this is the case and the violation is more severe than it was before, perhaps a poor decision was made the last time and should be avoided now. The designer may also look at the number of times area constraints have been relaxed in comparison to the number of times delay constraints have been relaxed. If the current constraint is an area constraint and there have been more relaxations of area constraints than delay constraints, it might be a good idea to choose redesign to resolve the current violation. These two cases are examples of the type of knowledge encoded in the HEO's. By using the HEO's to look at what it has done in the past, Viola has a better chance of avoiding past mistakes and making a better decision. When a cell is opened for design, a constraint violation history is associated with it. This history records any constraint violations handled by Viola, the severity rating of those violations and the actions taken by Viola to resolve them.

The knowledge contained in Viola's HEO's consists of a set of historical conditions and the course of action favoured by those conditions. The HEO's used by Viola

are listed in Table 5.2.

Table 5.2. Historical Evaluation Operators

HEO	If: (Conditions) Then Favour: (Recommendation) (VC = violated constraint, CC = competing constraint)	
	Conditions	Recommendation
HEO1	- the last VC was the same as the current VC - the previous ranking was lower - it was resolved by relaxation	redesign
HEO2	- the last VC was the same as the current VC - the previous ranking was lower - it was resolved by redesign	relaxation
HEO3	- a CC of the current VC has been violated in the past - it was resolved by redesign	relaxation
HEO4	- there have been more relaxations of CC types than of the current VC type	relaxation
HEO5	- there have been more relaxations of the current VC type than of CC types	redesign
HEO6	- the last violation of the current VC type was resolved by relaxation	redesign
HEO7	- the last violation of the current VC type was resolved by redesign	relaxation
HEO8	- the current VC has been violated before - the ranking was higher last time - it was resolved by relaxation	relaxation
HEO9	- the current VC has been violated before - the ranking was higher last time - it was resolved by redesign	redesign
HEO10	- the current VC has been violated before - the ranking was lower last time - it was resolved by relaxation	redesign
HEO11	- the current VC has been violated before - the ranking was lower last time - it was resolved by redesign	relaxation
HEO12	- the current VC is ranked higher than any previously violated CC	redesign

As with the SA0's, some of the HEO's are more important than others. HEO1 and HEO2 are more important than any others because they are intended to correct a mistake made on the last violation handling step. HEO's 4 to 7 refer only to constraint types rather than specific constraints; therefore, they are less important than the

other HEO's. For any HEO's registering a vote, the more recent the historical event they are referring to the more important they are. All of these distinctions are made by the meta-operators.

The active and applicable HEO's vote for relaxation or redesign in the same way as the SAO's. The result is that the relaxation list and the redesign list each contain a number of SAO and HEO voters. The next task to be performed is to decide which of the voters are the most important -- that is the task of the meta-operators.

5.3. Meta-Operators

The meta-operators take their name from the fact that they are operators that are created to reason about other operators. Their function is to compare operators in order to decide whether one is more important than another. The knowledge contained in Viola's meta-operators is listed below:

MetaOperator1:

> Any HEO whose intention is to indicate that the last decision made by Viola didn't work is more important than any other HEO.

MetaOperator2:

> An HEO referring to a more recent event in the history is more important than an HEO referring to an event in the more distant past.

MetaOperator3:

> An HEO that refers to a specific constraint is more important than an HEO that refers only to a constraint type.

MetaOperator4:

> An SAO that refers to the violated constraint and a competing constraint is more important than an SAO that refers only to the violated constraint.

MetaOperator5:

An SAO that refers to a single competing constraint is more important than an SAO that refers to multiple competing constraints.

MetaOperator6:

An HEO is more important than an SAO.

It is apparent from these meta-operators that they must have access to some knowledge about the SAO's and HEO's on which they operate. This additional knowledge is contained in a number of SAO and HEO variables called content description variables. With one exception noted below, these variables are set by the user when the operators are created. For SAO's, the *competitorsConsidered* variable may be set to 'nil', 'one' or 'multiple' so that meta-operators can make decisions based on how many competing constraints are referenced by an operator. For HEO's, there are a number of these content description variables as listed below:

correctionIntent

This variable indicates whether the intent of a HEO is to correct a mistake made on the previous violation handling step. It may be set to 'true' or 'false'.

singleEventReference

This variable indicates whether the HEO refers to a single historical event. It may be set to 'true' or 'false'.

typeReference

This variable indicates whether the HEO refers only to constraint types. It may be set to 'true' or 'false'.

historyIndex

This variable is not set by the user. It is set by the HEO itself when it is applied and indicates the point in the constraint violation history to which the HEO refers

(provided that *singleEventReference* is 'true' indicating that the HEO refers to a single historical event).

Some meta-operators apply to HEO's only, some apply to SAO's only and some apply to both. Each meta-operator contains knowledge about which types of operators it applies to. Viola uses the meta-operators to weed out less important voters from the redesign list and the relaxation list as shown in Figure 5.2.

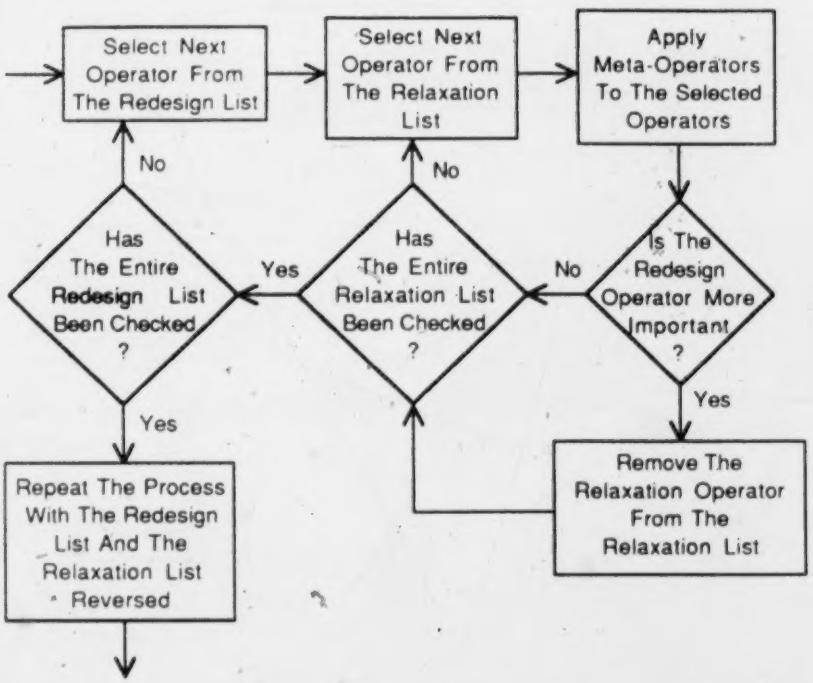

Select Next Operator From The Redesign List
Select Next Operator From The Relaxation List
Apply Meta-Operators To The Selected Operators
No
No
Has The Entire Redesign List Been Checked ?
Yes
Has The Entire Relaxation List Been Checked ?
No
Is The Redesign Operator More Important ?
Yes
Yes
Repeat The Process With The Redesign List And The Relaxation List Reversed
Remove The Relaxation Operator From The Relaxation List

Figure 5.2. Control Flow For Meta-Operator Application

First, any operators on the relaxation list that the meta-operators identify as less important than an operator on the redesign list are removed. Next, the process is repeated and any operators on the redesign list that the meta-opertors identify as less important than an operator on the relaxation list are removed. Once that has been done, all unimportant operators have been removed and the relax/redesign decision can

be made by comparing the sizes of the redesign list and the relaxation list. If the size of the redesign list is greater than the size of the relaxation list, the decision is made to redesign the cell; otherwise, the decision is made to relax the constraint. The proposed plan now consists of a statement of intent to relax the constraint or redesign the cell -- the role of the next stage is to refine and implement the plan.

Chapter 6. Viola's Plan Refinement And Implementation Stage

By the time Viola reaches the plan refinement and implementation stage, the relax/redesign decision has already been made for the current constraint violation. All that remains to be done is to decide exactly how to relax the constraint or redesign the cell. The control flow for this stage is presented in Figure 6.1.

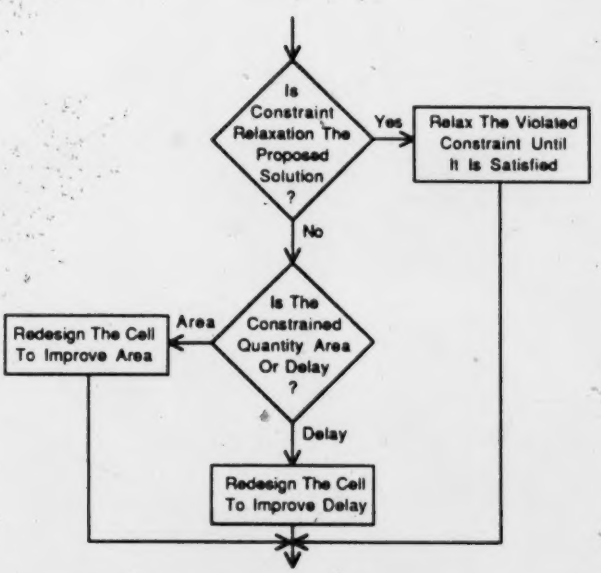

Is Constraint Relaxation The Proposed Solution ?
Yes
Relax The Violated Constraint Until It Is Satisfied
No
Is The Constrained Quantity Area Or Delay ?
Area
Redesign The Cell To Improve Area
Delay
Redesign The Cell To Improve Delay

Figure 6.1. Control Flow For Plan Refinement And Implementation

If the plan is to relax the violated constraint, there is no plan refinement required -- Viola may proceed directly with plan implementation and relax the constraint until it becomes satisfied. The idea of plan refinement really comes into play when the plan calls for cell redesign. A cell is redesigned by replacing one or more of its subcells with alternate implementations from STEM's cell library. Plan refinement selects subcells for replacement that will resolve the current constraint violation while having as little effect as possible on competing constraints.

Section 6.1 describes how constraint relaxation is implemented by Viola. In Section 6.2, the implementation of STEM's cell library is discussed. This library is

organized by functional category and each category is sorted by area and by the value of each delay variable declared for that cell type. Section 6.3 describes the plan refinement and implementation process for resolving an area constraint violation and section 6.4 discusses the different approach required to resolve a delay constraint violation.

6.1. Constraint Relaxation

Viola's objective in constraint relaxation is to relax the violated constraint until it is satisfied by the current values of its arguments. For plan implementation, a constraint is relaxed by its violation margin -- the minimum amount required to satisfy the constraint.

6.2. STEM's Cell Library

Since redesign consists of replacing subcells with functionally equivalent alternate implementations, a cell library was created for Viola. The cell library is organized by functional category. For example, the three inverters in Figure 6.2 are stored in a category called *Inverter*. Though this is a functional category, STEM performs no simulation to verify the function of the cells within the category -- this is left to the user.

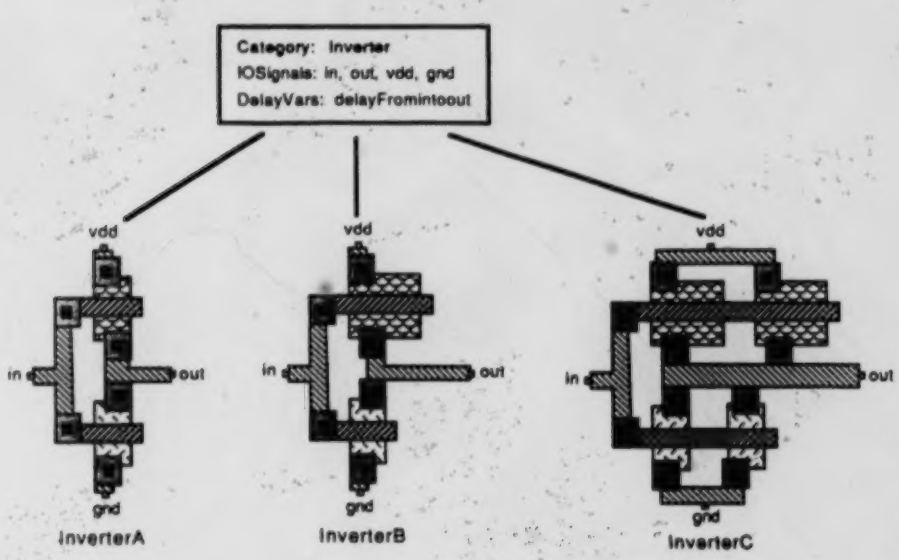


Figure 6.2. Functional Category Inverter

Cell redesign aims either to reduce area by choosing a smaller subcell implementation or to improve delay by choosing a faster subcell implementation; therefore, it was necessary to have the cell library organized in such a way that Viola can easily identify one implementation as smaller or faster than another. This was done by adding a *smallerAlternateCell* field to each area variable and a *fasterAlternateCell* field to each delay variable which are used to store pointers to smaller or faster implementations within a category. Each inverter in Figure 6.2 will have a *smallerAlternateCell* pointer associated with its area variable and a *fasterAlternateCell* pointer associated with its delay variables. Any time one of the inverter implementations is modified or a new implementation is added, category *Inverter* is automatically sorted by area and delay to update the pointers. The smallest implementation has its *smallerAlternateCell* pointer set to nil; likewise, the fastest implementation has its *fasterAl-*

ternateCell pointer set to nil. The result is that category *Inverter* becomes organized as shown in Figure 6.3 with all of the implementations linked by the *smallerAlternateCell* and *fasterAlternateCell* pointers.

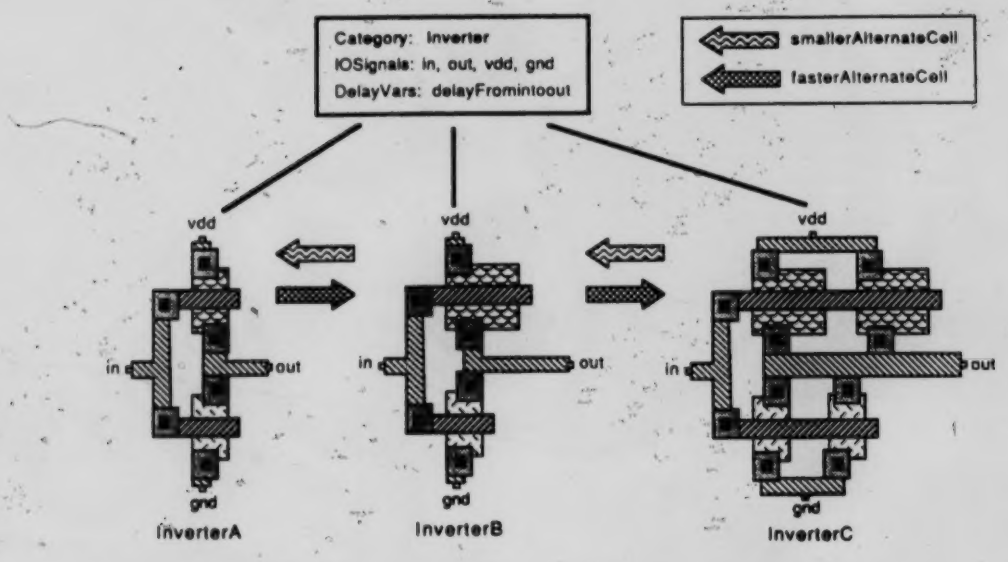


Figure 6.3. Functional Category Inverter After Sorting

These pointers are akin to the similarity links use by Shibahara in the CAA system [14]. The idea of similarity links is to direct search using links keyed on important differences between objects. For Viola, search for an alternate implementation begins at the current implementation. The difference between the desired alternate implementation and the current implementation is either a decrease in area or an increase in speed; thus, Viola's search is directed by similarity links keyed on smaller area *(smallerAlternateCell)* or greater speed *(fasterAlternateCell)*. This difference-directed search allows Viola to find appropriate implementations quickly.

6.3. Cell Redesign For Area Improvement

The first step in cell redesign is to determine which subcells are associated with the violated constraint (block A in Fig. 6.4). This group of subcells is called the critical subcell set (CSS) for the constraint. Since all subcells contribute to a cell's area, the CSS for an area constraint comprises all subcells. One or more members of this CSS will be chosen for replacement.

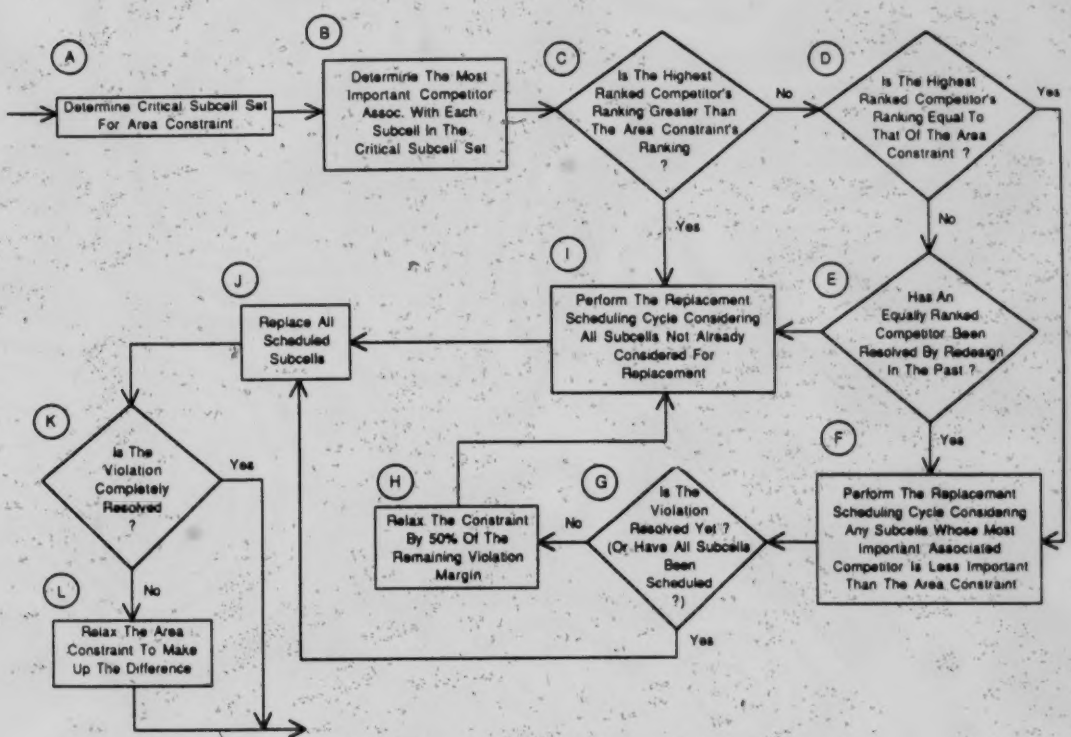

A
Determine Critical Subcell Set For Area Constraint
B
Determine The Most Important Competitor Assoc. With Each Subcell In The Critical Subcell Set
C
Is The Highest Ranked Competitor's Ranking Greater Than The Area Constraint's Ranking ?
No
D
Is The Highest Ranked Competitor's Ranking Equal To That Of The Area Constraint ?
Yes
No
Yes
I
Perform The Replacement Scheduling Cycle Considering All Subcells Not Already Considered For Replacement
E
Has An Equally Ranked Competitor Been Resolved By Redesign In The Past ?
J
Replace All Scheduled Subcells
Yes
F
Perform The Replacement Scheduling Cycle Considering Any Subcells Whose Most Important Associated Competitor Is Less Important Than The Area Constraint
G
Is The Violation Resolved Yet ? (Or Have All Subcells Been Scheduled ?)
No
H
Relax The Constraint By 50% Of The Remaining Violation Margin
Yes
K
Is The Violation Completely Resolved ?
Yes
No
L
Relax The Area Constraint To Make Up The Difference

Figure 6.4. Cell Redesign Control Flow For Area Constraints

The intelligent selection of subcells for replacement requires a certain amount of information about each subcell in the CSS. This information is recorded by assigning each subcell to an object called a *RedesignInfoRecord* that has the information fields

shown in Figure 6.5.

subcellInstance	- subcell associated with this RedesignInfoRecord
improvedVars	- subcell variables associated with the violated constraint
mostImportantCompetitor	- highest ranked competing constraint associated with subcellInstance
competitorRanking	- ranking of mostImportantCompetitor
worsenedVars	- subcell variables associated with mostImportantCompetitor
alternateImplementation	- selected alternate implementation for subcell replacement
improvement	- percentage improvement in the violated constraint
worstEffect	- percentage adverse effect on mostImportantCompetitor
netBenefit	- (improvement - worstEffect)
numCriticalPaths	- used for delay constraints only

Figure 6.5. A RedesignInfoRecord

The only fields specified initially in the *RedesignInfoRecord* are the *subcellInstance* field and the *improvedVars* field. *improvedVars* stores the name(s) of the subcell variable(s) that contribute to the cell variable constrained by the violated constraint. For area, the subcell variables are always named *area* but for delay, the subcell delay variable may, in some cases, be any one of several delay variables. By comparing the value of the *improvedVars* for an alternate implementation and for the current implementation, Viola can determine how much improvement is possible with a particular replacement. The remaining fields of the *RedesignInfoRecord* are filled in as the redesign process progresses.

To determine the adverse effect of subcell replacement, Viola determines the highest ranked competing constraint associated with each of the subcells in the CSS (block B in Fig. 6.4). A subcell is said to be associated with a competing constraint if it is included in the CSS of that constraint; consequently, Viola must determine the

CSS of each competing constraint and check each subcell for membership in those CSS's. When the highest ranked competitor associated with a subcell is found, it is recorded in that subcell's *RedesignInfoRecord* along with its ranking. At this point, the subcell variables associated with the *mostImportantCompetitor* are recorded in the *worsenedVars* field. These variables are used to calculate the adverse effect of a replacement on the competitor.

Blocks C through G of Figure 6.4 were added to the plan refinement and implementation stage as a heuristic compensation for a problem encountered during testing of Viola. Since Viola operates on the principle of resolving one constraint violation at a time, a situation where the violation list contained two equally ranked competing constraint violations was resulting in one of two scenarios:

1. (a) One of the constraint violations would be resolved by redesign resulting in a large detrimental effect on the other constraint.

   (b) The second constraint would be relaxed.

2. (a) Same as step (a) in 1.

   (b) The second constraint violation would be resolved by redesign; thereby, undoing much of the redesign done in step (a).

   (c) This goes back and forth until one constraint is relaxed.

The result in both of these scenarios was that one constraint would be satisfied without relaxation (or as near as possible) and the other would be relaxed by a large amount, It was felt that it would be more desirable to achieve some middle of the road solution where both constraints would be relaxed by a lesser amount. The idea is to balance the improvement in the constraint currently being handled with the adverse effect on its competitor by setting a redesign target of only 50% of the violation margin and making up the other 50% by relaxation. The solution used is to check for the

condition where the highest ranked competitor's ranking is equal to that of the area constraint (block D in Fig. 6.4) or where an equally ranked competitor was resolved by redesign in the past (block E in Fig. 6.4). If either of these conditions exist, Viola proceeds with a "50% relaxation heuristic" (blocks F through H). Since it may be possible to partially or completely resolve the area constraint violation by replacing subcells whose most important associated competitor is less important than the area constraint, the subcell replacement scheduling cycle is performed (block F) before applying the 50% relaxation. If this is sufficient to completely satisfy the area constraint violation, no 50% relaxation is performed. If, however, the area constraint is still violated, it is relaxed by 50% of the remaining violation margin before proceeding with redesign. The result is that over two or more violation handling passes, the two constraints approach a similar level of relaxation rather than oscillating back and forth until one constraint is relaxed by a large amount as before.

As an example, consider the constraints depicted in Figure 6.6. Figure 6.6 (a) shows the results of three violation handling passes without the 50% relaxation heuristic and Figure 6.6 (b) shows the results of three violation handling passes using the 50% relaxation heuristic. The total relaxation required in (b) is about the same (21%) as in (a) (23%) but it is split between the two constraints producing a more balanced solution. The resultant solution in many cases may not be optimal; however, the quality of the solution is improved significantly without adding a great deal of computational complexity.

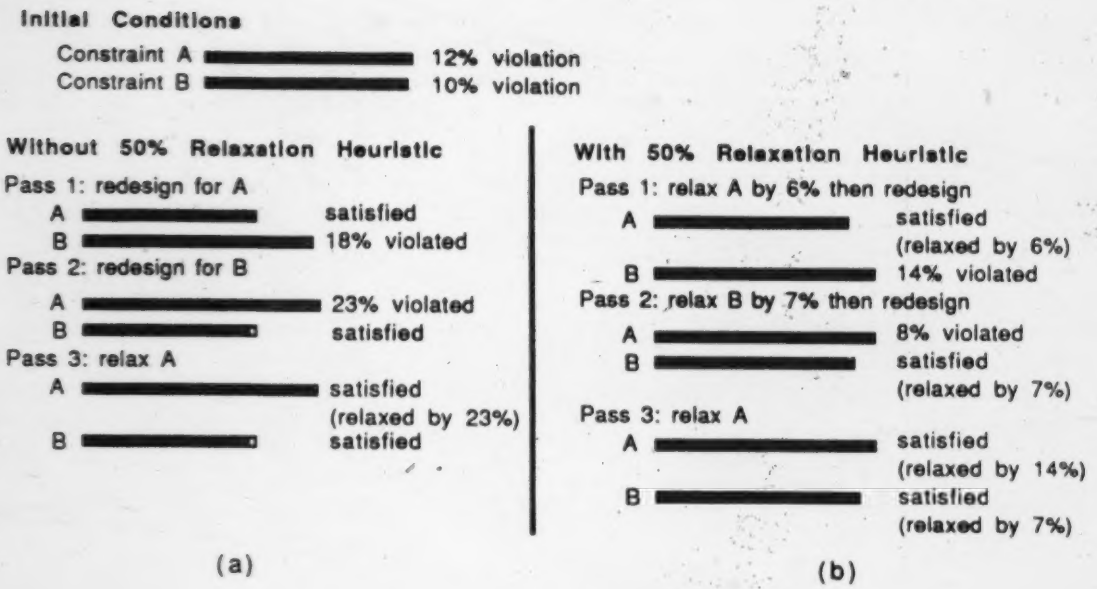


Figure 6.6. A Violation Handling Example

Figure 6.7 depicts the subcell replacement scheduling cycle for an area constraint violation. The replacement scheduling cycle may be described as a heuristic search where the starting point is the condition where the area constraint is violated, the goal is the condition where the area constraint is no longer violated, each node of the search space is a partial solution and moves from node to node are made by selecting a subcell for replacement. The heuristic method used to select the next move is implemented by the two criteria used to select the best subcell replacement.

The first step in the cycle is to select an alternate implementation for each subcell that will come as close as possible to resolving the area constraint violation. For block F in Figure 6.4, only subcells whose most important associated competitor is less important than the area constraint being handled are considered for replacement. For block I in Figure 6.4, all subcells not already scheduled for replacement are con-

sidered. For each subcell, its alternate implementation, if any was found, is recorded in the *alternateImplementation* field of the *RedesignInfoRecord*. The improvement in the area constraint is calculated for each potential replacement by subtracting the value of the *improvedVars* for the alternate implementation from the value for the current implementation. The result is stored as a percentage of the constrained area value in the *improvement* field of the *RedesignInfoRecord*.

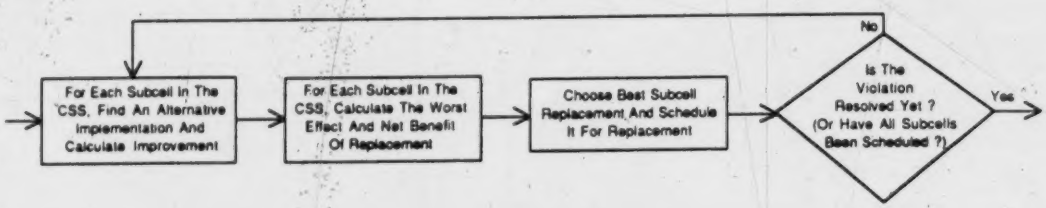

For Each Subcell In The CSS, Find An Alternative Implementation And Calculate Improvement
For Each Subcell In The CSS, Calculate The Worst Effect And Net Benefit Of Replacement
Choose Best Subcell Replacement And Schedule It For Replacement
Is The Violation Resolved Yet ? (Or Have All Subcells Been Scheduled ?)
No
Yes

Figure 6.7. Subcell Replacement Scheduling Cycle

To be able to choose one subcell replacement over the others it is necessary to consider more than just the potential improvement -- the detrimental effect on competing constraints must also be considered. To consider adverse effects on all competing constraints may, however, become very time consuming in the case where there are several competitors. A computationally expedient approximation used by Viola is to consider the effect only on the highest ranked competitor recorded earlier in each subcell's *RedesignInfoRecord*. The detrimental effect on the competitor is calculated by comparing the values of the *worsenedVars* for the current subcell implementation and for the selected alternate implementation. This difference is recorded in the *worstEffect* field as a percentage of the constrained value of the competitor. A measure of the net benefit of a subcell replacement is now available by subtracting the *worstEffect* from the *improvement*. This measure is stored in the *netBenefit* field.

The final step in the subcell replacement scheduling cycle is to select the best subcell replacement. The selection is made on the basis of two factors -- the ranking of the subcell's most important associated competitor and the net benefit of the replacement (in that order). Once the best subcell replacement has been selected, it is scheduled by placing its *RedesignInfoRecord* on the replacement schedule. The subcell replacement scheduling cycle continues until either the scheduled replacements are sufficient to resolve the constraint violation or all possible replacements have been scheduled.

Once the replacement schedule is complete, all of the scheduled subcell replacements are implemented by removing the required subcells from the cell and inserting their alternate implementations. At present, STEM has no automatic connection and routing facility; therefore, when the scheduled subcells have been replaced, control is returned to the user so that he can reestablish the necessary connections.

In some cases, a constraint may be violated by such a wide margin that replacing all possible subcells isn't enough to resolve it. It is for these cases that block L has been included in Figure 6.4. In order to ensure that the constraint violation is competely resolved in these cases, Viola takes up any residual violation margin by relaxing the violated constraint at the end of the redesign process.

6.4. Cell Redesign For Delay Improvement

The plan refinement and implementation process for resolving delay constraint violations (Fig. 6.8) is very similar to that for resolving area constraint violations. There are, however, a few differences in the process that arise from a fundamental difference between cell area and cell delay -- a value for cell area has only one source, the total area of all subcells; whereas, a value for propagation delay through a cell arises from the longest of potentially many delay paths. In order to resolve a delay

constraint violation, it is necessary to reduce the delay of any paths that are long enough to violate the constraint (violated paths).

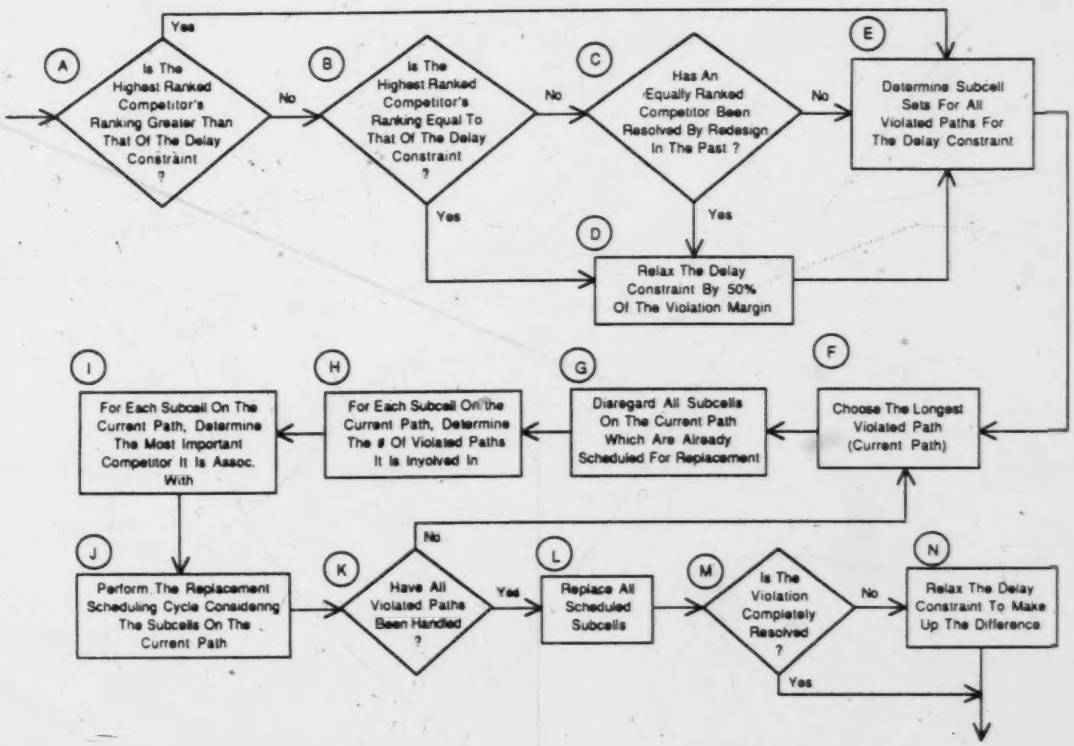

A
Is The Highest Ranked Competitor's Ranking Greater Than That Of The Delay Constraint ?
Yes
No
B
Is The Highest Ranked Competitor's Ranking Equal To That Of The Delay Constraint ?
No
Yes
C
Has An Equally Ranked Competitor Been Resolved By Redesign In The Past ?
No
Yes
D
Relax The Delay Constraint By 50% Of The Violation Margin
E
Determine Subcell Sets For All Violated Paths For The Delay Constraint
F
Choose The Longest Violated Path (Current Path)
G
Disregard All Subcells On The Current Path Which Are Already Scheduled For Replacement
H
For Each Subcell On the Current Path, Determine The # Of Violated Paths It Is Involved In
I
For Each Subcell On The Current Path, Determine The Most Important Competitor It Is Assoc. With
J
Perform The Replacement Scheduling Cycle Considering The Subcells On The Current Path
K
Have All Violated Paths Been Handled ?
No
Yes
L
Replace All Scheduled Subcells
M
Is The Violation Completely Resolved ?
No
Yes
N
Relax The Delay Constraint To Make Up The Difference

Figure 6.8. Cell Redesign Control Flow For Delay Constraints

The first difference for delay constraints is the way that the concept of critical subcell set (CSS) is used. In resolving an area constraint violation as described in the previous section, the CSS of all competing constraints (i.e. all delay constraints on the cell) had to be determined in order to decide whether or not a subcell was associated with that competitor. For a delay constraint, the CSS comprises all subcells contained in paths that are within 10% of being violated. The factor of 10% is used so that any subcells whose replacement would likely violate the delay constraint are included. If

the delay constraint has no paths that are close to violation, the CSS just comprises the subcells on the longest delay path. This CSS is used only for the purpose described above -- as shown in Figure 6.8, each of the violated paths is considered separately in the replacement scheduling process; therefore, a separate subcell set is used for resolving each violated path.

The 50% relaxation heuristic is applied in a simpler way for delay constraints than for area constraints -- since all subcells are associated with any area constraint on the cell, there is no potential to improve delay without affecting area. It is therefore unnecessary to perform the subcell replacement scheduling cycle before performing the 50% relaxation.

The resolution of the violated paths starts with the longest violated path and works down to the shorter paths. In this way, the shorter paths are frequently resolved as a side effect of resolving the longer paths. Scheduling subcell replacements to resolve a single path is very similar to the process of scheduling subcell replacements to resolve an area constraint violation. The subcell replacement scheduling cycle is the same as the replacement scheduling cycle for area constraint violations. The difference arises in blocks G and H of Figure 6.8 which take into account the fact that a single subcell may take part in more than one of the violated delay paths. If a subcell has already been replaced to resolve a previous violated path, it makes no sense to consider it for replacement again in resolving another violated path; consequently, block F removes any subcells from consideration that have already been scheduled for replacement. Block H adds a value to the *numCriticalPaths* field of the subcell's *RedesignInfoRecord*. This value specifies the number of violated paths that the subcell is involved in and provides an additional decision criterion for selecting a subcell for replacement. The idea here is that it is better to start by replacing subcells common to several delay paths because the potential for resolving more than one violated path at

once is thereby increased. The selection of the best subcell replacement for delay constraints takes three factors into consideration -- the number of violated paths the subcell is involved in, the net benefit of the replacement and the ranking of the subcell's highest ranked associated competitor (in that order).

Plan refinement and implementation for delay constraint satisfaction differs from the process for area constraint satisfaction in that the subcell replacement scheduling cycle is executed for each violated path of the delay constraint. The replacement scheduling cycle for a violated path may be described as a heuristic search where the starting point is the condition where the path is violated, the goal is the condition where the path is no longer violated, each node of the search space is a partial solution and moves from node to node are made by selecting a subcell for replacement. The heuristic method used to select the next move is implemented by the three criteria used to select the best subcell replacement.

Chapter 7. Discussion Of Results

This chapter discusses results obtained with Viola on seven constraint satisfaction problems. Viola's solutions are examined and compared to results obtained by hand. Section 7.1 describes the subcell library and the test cell used for the problems.

There are two major types of problems. The first type occurs where there is one overriding constraint that is more important than any of the other constraints on a cell -- Section 7.2 will look at three problems of this type. The second type of problem occurs where there is not a single overriding constraint. In many ways, the second type of problem is more difficult to resolve. Section 7.3 will look at four problems of this type.

7.1. The Subcell Library And The Test Cell

The subcell library used for the problems discussed in this chapter is depicted symbolically in Figure 7.1. It consists of four inverter implementations, three two-input NAND gate implementations and three three-input NAND gate implementations. The area and delay values for each implementation in the library are shown. SPICE [20] was used to calculate the unloaded delay values shown for the library cells because STEM's delay calculation doesn't take transistor dimensions into consideration and the area and delay of the library cells was varied by transistor sizing. As discussed in Section 3.4, STEM adds additional RC delay to account for loading of circuit subcells.

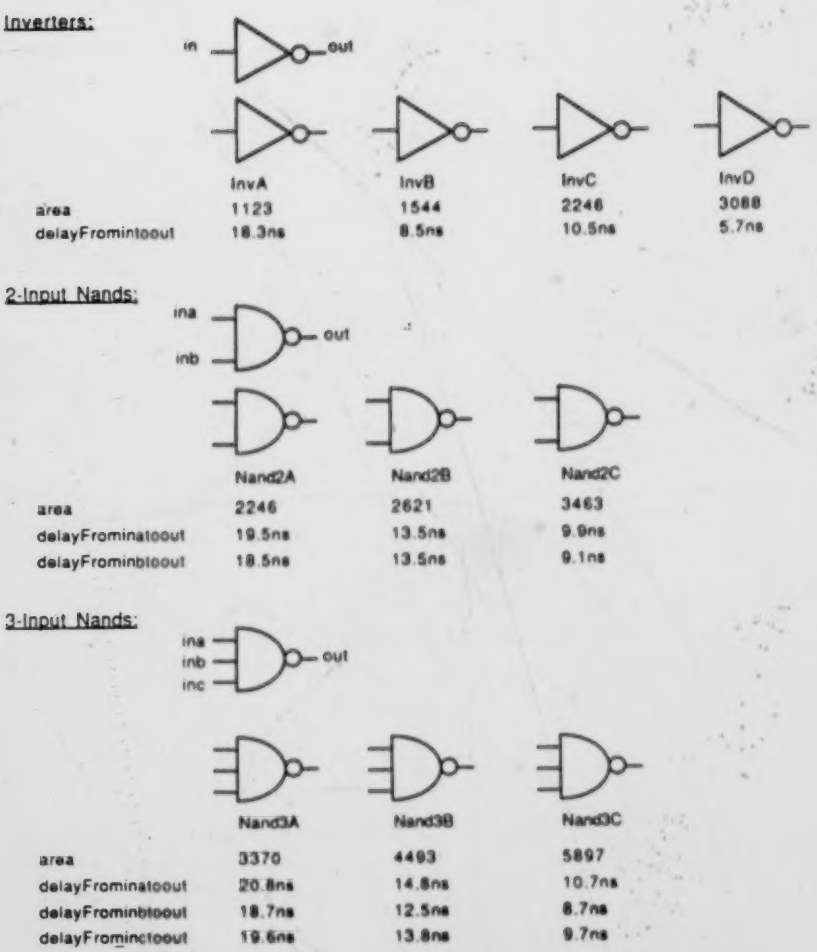


Figure 7.1. The Subcell Library

In order to properly demonstrate Viola's capabilities, a test cell with several delay variables and a variety of delay paths was designed and is presented in Figure 7.2. For this cell, there are four variables that may be constrained by the user -- area, delayFrominatoout, delayFrominbtoout and delayFrominctoout. This test cell has a large enough number of subcells to permit many possible ways of redesigning the cell and has several delay paths of varying length for each delay variable.

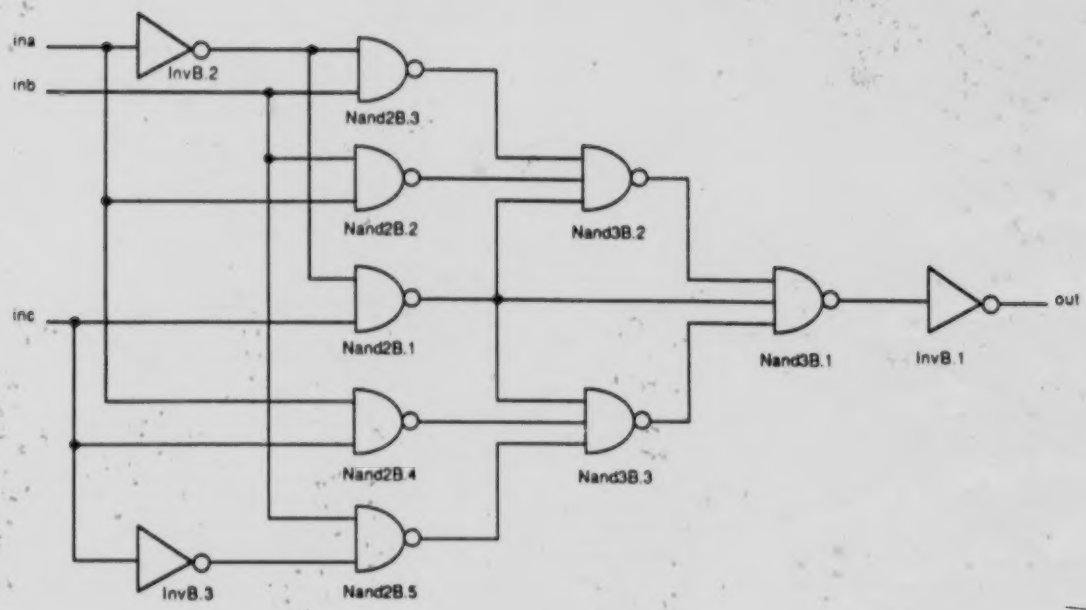


Figure 7.2. TestCellA

For each of the constraint satisfaction problems described in the next two sections, Viola was presented with a set of two or more constraints (some violated and some not) and the complete circuit for TestCellA as shown in Figure 7.2. The discussion looks at the steps taken by Viola to resolve the constraint violations and the reasons for taking those steps. Since Viola's knowledge base was derived from the author's problem solving knowledge, the actions taken and the results obtained by Viola are evaluated by comparing them with the author's hand solutions. This comparison is valid because the hand solutions were generated using an exhaustive parallel constraint satisfaction approach rather than Viola's approach. This parallel approach was avoided in Viola's design to prevent combinatorial problems in large circuits; as a result, the quality of the hand solutions will degenerate more quickly with increasing circuit complexity than will the quality of Viola's solutions.

7.2. Constraint Satisfaction Problems With A Single Overriding Constraint

For problems where there is one constraint that is more important than any of the others, Viola's problem solving strategy is based on the following premise:

> "Satisfy the most important constraint if at all possible. Any adverse effect on less important constraints is a secondary concern."

Problems of this type will be referred to as "type I problems". The initial conditions for the three problems examined in this section are presented in Table 7.1. In Test Cases 1 and 2, the overriding constraint is already violated when Viola is called upon. In Test Case 3, the overriding constraint is not yet violated but is in danger of being violated by any actions taken to resolve the violated competing constraints.

Table 7.1. Initial Conditions For Type I problems

	Constrained Variables (VM = Violation Margin, Imp = Importance)							
	area		delayFrominatoout		delayFrominbtoout		delayFrominctoout	
	VM	Imp	VM	Imp	VM	Imp	VM	Imp
Test Case 1	+6.8%	VI	-5.4%	I	-3.4%	I	-2.4%	I
Test Case 2	+1.5%	I	+70.2%	VI	+35.2%	I	+56.2%	NI
Test Case 3	-5.6%	VI	+6.4%	I	+12.7%	I	+4.1%	I

Transcripts of the actions taken by Viola in resolving these problems are presented in Appendix I. In order to properly evaluate Viola's solutions, a number of solution quality metrics have been used. The first of these metrics is the percentage relaxation needed to satisfy each constraint (PRTS for short). The PRTS is measured as a percentage of the original constrained value. In any problem, it is most desirable to satisfy each constraint without any relaxation at all (PRTS = 0); however, this is not usually possible.

The next metric used to evaluate solution quality is the number of subcells in the final version of the cell that are different from the original cell. It is desirable to resolve a problem by making as little change to the designer's original circuit as possible. An excessive number of different subcells can indicate either that Viola chose to redesign the cell too often or that its selection of subcells for replacement was inefficient.

The final metric involves a comparison of the total number of subcell replacements made in the violation handling process with the number of different subcells in the final version of the cell. The idea here is that any redesign step may involve the replacement of subcells that have already been replaced in previous redesign steps. This measure of the number of reversals of previous replacements gives an indication of the cost of Viola's methodology of handling constraint violations one at a time. Since a human designer will usually consider all constraints in parallel, a manual solution will seldom involve any reversals. As stated earlier, however, this will be true only for relatively simple problems.

The resultant values of all of these solution quality metrics for the type I problems are presented in Table 7.2 both for Viola's solution and for the author's hand solution. Each of the problems is discussed individually in the following subsections; then, the results for all type I problems are summarized.

Table 7.2. Results For Type I problems

	PRTS, area		PRTS, delayFrominatoout		PRTS, delayFrominbtoout		PRTS, delayFrominctoout		Number Of Different Succells		Total Number Of Replacements		Number Of Reversals	
	Viola	Hand	Viola	Hand	Viola	Hand	Viola	Hand	Viola	Hand	Viola	Hand	Viola	Hand
Test Case 1	0	0	1.2	1.2	10.8	3.7	11.1	4.9	4	4	4	4	0	0
Test Case 2	30.6	30.6	35.4	35.4	9.2	9.2	34.2	34.2	7	7	7	7	0	0
Test Case 3	0	0	9.8	1.3	7.3	5.8	0	0	5	1	7	1	1	0

7.2.1. Test Case 1

This test case involves satisfying a very important area constraint in the presence of three less important delay constraints. Initially, the area constraint is violated by 6.8% and the delay constraints are all satisfied but close to violation. The goal here is to satisfy the area constraint while causing the delay constraints to be violated by as little as possible.

In Viola's solution, the area constraint was resolved with a PRTS of 0; hence, the primary goal of satisfying the original area constraint was met. The secondary goal of adversely affecting the delay constraints as little as possible was not as well met by Viola as in the hand solution. In the hand solution, each of the delay constraints was violated by a small amount and the adverse effect was spread evenly over the three constraints. Viola didn't distribute the adverse effect as well. In choosing subcell replacements, Viola only considers the effect on the highest ranked competitor associated with each subcell. As a result, a subcell replacement that adversely affects more than one constraint but has a small adverse effect on its highest ranked competitor may be chosen over a replacement that affects only one constraint but has a slightly larger

effect on it. This inefficiency in replacement selection is the price to be paid for the computational advantage of only having to track effects on a single competitor for each potential subcell replacement.

The other solution quality metrics show no difference between Viola's solution and the hand solution. Viola chose to replace the same number of subcells as in the hand solution, but, as explained above, Viola's replacements were chosen with a narrower focus and resulted in a poorer solution.

Viola's solution to the problem of test case 1 is acceptable as it met the primary goal and fell short of the optimal solution (assuming the hand solution to be optimal) only by a small amount.

7.2.2 Test Case 2

Test case 2 involves resolving a very important delay constraint violation in the presence of a less important area constraint. There are also two less important delay constraints. The primary difference from test case 1 is that there is only one constraint competing with the very important delay constraint; hence, there is no error introduced by considering only the highest ranked competitor associated with each subcell.

For this test case, the hand solution and Viola's solution are identical. The primary goal of satisfying the very important delay constraint was not met in either case because the violation margin was too large. In both solutions, all subcells on the critical delay paths were replaced by their fastest alternate implementations. Once that was done, there was no further improvement available and the remaining 35.4% was made up by relaxation.

7.2.3 Test Case 3

As mentioned earlier, this test case is different from the first two test cases in that the very important constraint is not initially violated. The problem here is to try to resolve three important delay constraint violations in the presence of a very important area constraint that is close to being violated.

For this problem, Viola achieves results very similar to the hand solution; however, the means of achieving those results is much different. Viola's first step was to select the most severe constraint violation and resolve it. Initially, the constraint on delayFrominbtoout was the most severely violated. Viola chose to resolve this violation by redesign. This redesign step, however, caused the more important area constraint to be violated. Since Viola's ranking scheme is set up to consider any violated constraint more significant than any non-violated constraint, the area constraint, though more important, was considered less significant than the delay constraints until it was violated. As soon as the area constraint was violated, it became the focus of Viola's attention.

To resolve the area constraint violation, Viola performed a second redesign step. It is this use of two redesign steps that accounts for the difference between Viola's solution and the hand solution. Viola's first redesign step replaced three subcells with faster implementations; then, Viola's second redesign step reversed one of the initial replacements and replaced three other subcells with smaller implementations. In generating the hand solution, it was possible to identify that what had to be done was to improve the delay constraints as much as possible without causing the area constraint to become violated. As it turned out, a single subcell replacement was sufficient to achieve this goal.

For Viola to predict that its actions would cause a more severe constraint violation, it would be necessary to reevaluate constraint rankings and reapply precedence operators after each subcell replacement. This would slow the system down considerably for all redesign steps in order to provide a more efficient solution for problems like test case 3. Since problems of this type are a minority, the tradeoff is not justifiable.

Although Viola's means of achieving its solution were less efficient than the hand solution, the end result was near enough to the hand solution to be acceptable.

7.2.4 Summary Of Results For Type I problems

For all of the type I problems, the primary goal of satisfying the overriding constraint without relaxation was achieved. There were, however, two problems resulting from Viola's problem solving approach.

The first problem with Viola's approach is its consideration of adverse effects only on the highest ranked competing constraint associated with each subcell. This resulted in a less than optimal distribution of adverse effects over competing constraints. This narrowing of Viola's focus also provides a benefit to offset the problem -- considering a single competitor for each subcell reduces the computing time required for subcell replacement selection. For a small cell like TestCellA, this speed-up is marginal; however, for a large cell with hundreds to thousands of subcells and dozens of constraints, the benefit could become significant.

The second problem is Viola's inability to determine that a redesign step may cause a more severe problem than the one it is trying to fix. Viola responds properly and resolves this more severe problem, but that introduces a second redesign step to the violation handling process; thus, reducing solution efficiency. Viola has the ability

to reason about constraint violation severity with its precedence operators. This reasoning process, however, is not suited to incremental comparison of constraints within the subcell replacement cycle. To do this would require recalculation of the delay network, re-ranking of all constraints and reapplication of the precedence operators after each subcell replacement. The added computation would be greater than that introduced by the extra redesign step in the first place.

Despite these deviations from optimal problem solving, Viola achieved acceptable solutions for all type I problems.

7.3. Violation Handling With Equally Important Constraints

For this second type of problems (type II for short), there is no single overriding constraint so the approach to problem solving is different. The problem solving strategy for type II problems can be stated as:

> "In resolving the violated constraints, try to balance the improvement in a violated constraint with any adverse effects on its equally important competitors."

The 50% relaxation heuristic plays a fundamental role in this balancing of adverse effects.

The initial conditions for four type II problems are presented in Table 7.3. The results for these problems are presented in Table 7.4 and discussed in the following subsections. A summary of the results obtained for all type II problems is also presented. Transcripts of the actions taken by Viola in solving these problems are included in Appendix I.

Table 7.3. Initial Conditions For Type II problems

	Constrained Variables (VM = Violation Margin, Imp = Importance)							
	area		delayFrominatoout		delayFrominbtoout		delayFrominctoout	
	VM	Imp	VM	Imp	VM	Imp	VM	Imp
Test Case 4	-18.8%	I	+41.8%	I	/	/	/	/
Test Case 5	+4.1%	I	-5.4%	I	/	/	/	/
Test Case 6	+1.5%	I	+6.4%	I	/	/	/	/
Test Case 7	+1.5%	I	+21.6%	I	+4.0%	I	+11.8%	I

Table 7.4. Results For Type II problems

	PRTS, area		PRTS, delayFrominatoout		PRTS, delayFrominbtoout		PRTS, delayFrominctoout		Number Of Different Subcells		Total Number Of Replacements		Number Of Reversals	
	Viola	Hand	Viola	Hand	Viola	Hand	Viola	Hand	Viola	Hand	Viola	Hand	Viola	Hand
Test Case 4	4.5	4.5	12.8	12.8	/	/	/	/	7	7	7	7	0	0
Test Case 5	0	0	0	0	/	/	/	/	4	4	4	4	0	0
Test Case 6	0	1.0	3.2	1.3	/	/	/	/	5	5	5	5	0	0
Test Case 7	12.4	5.8	15.5	7.7	1.1	1.1	1.7	5.7	8	8	10	8	3	0

7.3.1. Test Case 4

In this test case, Viola was required to resolve a severely violated delay constraint in the presence of an equally important (though initially lower ranked) area constraint. Viola's initial action was to redesign the cell to improve the delay from ina to out.

Because the violation margin of the delay constraint was very large, it could not be completely resolved -- the delay paths were reduced as much as possible and the remaining 12.8% was made up by relaxing the constraint. Viola's next action was to relax the area constraint.

The result was as well balanced as possible given that the delay constraint couldn't be improved any further.

7.3.2. Test Case 5

For test case 5, Viola had to resolve an area constraint violation with as little adverse effect as possible on an equally important delay constraint that was close to being violated.

Both Viola's solution and the hand solution obtained ideal results for this test case, satisfying both constraints without any need for relaxation. In selecting subcells for replacement to improve the area, Viola first chose subcells that were not included in the critical subcell set of the delay constraint. For this problem, it was possible for Viola to completely resolve the area constraint without adversely affecting the delay constraint.

7.3.3. Test Case 6

This test case is an example of a problem where the 50% relaxation heuristic was used to achieve a balance between two equally ranked competing constraints. Viola's attention was first focussed on the delay constraint. In redesigning the cell to improve this delay, Viola identified that the highest ranked competitor of the delay constraint was equally ranked. To avoid having too large an adverse effect on this competing

area constraint, Viola chose to relax the delay constraint by 50% of the violation margin before carrying out the redesign. Viola's next step was to focus attention on the area constraint -- it was satisfied by redesign without any adverse effect on the delay constraint.

The end result achieved by Viola was that the area constraint was satisfied without relaxation and the delay constraint was satisifed with PRTS = 3.2%. This solution is not as well balanced as that achieved by hand. The imbalance results from the approximate nature of the 50% relaxation heuristic.

For this test case, Viola was once again able to achieve an acceptable, non-optimal solution.

7.3.4. Test Case 7

Test case 7 was the most complex of the type II problems. It involved the resolution of four equally ranked constraint violations -- three delay constraints and an area constraint. Though the constraints are equally ranked initially, there is a significant difference in actual violation margin. This is an example of "quantization error" introduced by using discrete violation margin intervals to rank constraints. Since the violation margin intervals used for testing Viola were 25% wide, there is a maximum of about 25% quantization error. In other words, two constraints whose violation margins are up to 25% different may be ranked the same.

Viola's strategy of handling constraint violations one at a time resulted in three redesign steps being performed, applying the 50% relaxation heuristic each time. This helped balance the opposing constraints; however, it also resulted in more subcell replacements than in the hand solution with three of them reversing earlier replacements. The relaxations were not as evenly distributed as in the hand solution -- a

result, once again, of the approximate nature of the 50% relaxation heuristic.

Despite the unevenness of the distribution, Viola's solution to this problem is acceptable with an average PRTS of 7.7% compared to 5.0% for the hand solution.

7.3.5. Summary Of Results For Type II problems

For all of the type II problems, Viola achieved acceptable results and achieved optimal results for two test cases. The non-optimal results in the other two test cases stem from three sources.

The first source of error is the 50% relaxation heuristic. This heuristic was added to the plan refinement and implementation stage to help improve the balance of the results of redesign without adding a great deal of extra computation. It achieves that end in producing results that are better balanced, but its computational simplicity results in non-optimal solutions.

The imbalance in Viola's results also received a contribution from the quantization error inherent in the constraint ranking system. This error is introduced by ranking constraints equally when there is actually a difference in their violation margins. By introducing additional constraint rankings, this error could be reduced.

As with the type I problems, Viola's need to handle constraint violations one at a time rather than in parallel resulted in extra subcell replacements; however, these extra subcell replacements only affected the efficiency of obtaining the solution, not the quality of the results.

Chapter 8. Conclusion

This chapter begins with a brief restatement of Viola's approach to the problem of constraint violation handling in IC design. The strengths and limitations of this approach are then discussed. The chapter concludes with suggestions for future research.

8.1. Handling Constraint Violations In IC Design

For a design environment that has competing objectives, it is necessary to reason about design tradeoffs when resolving constraint violations. Integrated circuit design is an example of such a design environment. In integrated circuit design, minimizing area and minimizing delay are competing objectives -- reducing the area of a cell will generally increase the propagation delay of signals passing through the cell and vice versa. Viola addressed this need to reason about area-delay tradeoffs in IC designs while resolving area and delay constraint violations. Its strategy was realized as three stages -- the focus of attention stage, the plan proposal stage and the plan refinement and implementation stage.

The focus of attention stage selects one of the violated constraints to resolve. To select the most severe constraint violation, Viola uses a severity rating scheme and a group of precedence operators. A numerical severity rating based on importance and extent of violation is assigned to each violated constraint and each competing constraint. The precedence operators then take these severity ratings and apply knowledge about how the presence and severity of competing constraints impacts the overall severity of a constraint violation. This harrowing of Viola's focus early in the process reduces the computation required later.

To resolve a constraint violation, Viola may either relax the violated constraint or redesign the IC cell. This decision is made in the plan proposal stage. Viola uses two sets of operators to "vote" for either constraint relaxation or cell redesign -- severity assessment operators representing knowledge about constraint violation severity and historical evaluation operators representing knowledge about the history of constraint violations and violation handling activities. The voting process is controlled by a group of meta-operators that represent knowledge about characteristics that make one severity assessment operator or historical evaluation operator more important than another.

Finally, in the plan refinement and implementation stage, the proposed plan is refined into a set of specific actions for Viola to take -- either a specific constraint relaxation, specific subcell replacements, or a combination of the two.

These three stages constitute Viola's handling of a single constraint violation. By reevaluating the constraints and repeating this process in a cyclic fashion until all constraints are satisfied, Viola produces a design that meets the original design objectives or balances adverse effects on those objectives.

8.2. Discussion

The primary objectives in Viola's design were:

(a) The violation handling strategy should consider constraint violation severity and the presence of competing constraints.

(b) The strategy should consider the violation handling history of the design.

(c) The strategy should be as flexible and explicit as possible.

(d) There should be a clear separation of domain-dependent and domain-independent

knowledge to allow transfer to other domains.

(e) The system should be computationally efficient to be useful for large designs.

(f) The system should enable an inexperienced designer to produce better designs in a shorter time and should reduce the time required for an experienced designer to produce good designs.

The following sections discuss the way that Viola met these objectives. Section 8.2.1 discusses the strong points of Viola's approach and section 8.2.2 looks at ways in which Viola could be improved.

8.2.1 Benefits Of Viola's Approach

In previous systems such as Borning's [1] ThingLab system and Fox's [4] ISIS system, the only way of reacting to differences in constraint violation severity was to try different predefined corrective actions until one of them satisfied the constraint. Viola improves on this approach by using a group of severity assessment operators (SAO's). Input from the SAO's makes Viola's decision-making process sensitive to the severity of the violated constraint and its competitors as set out in objective (a). By looking at both the violated constraint and its competitors, the SAO's balance Viola's decision between the need to improve the violated constraint and the need to avoid adversely affecting other constraints. In this way, Viola uses the concept of competing constraints to predict the potential adverse effects of its decisions -- something that was not done in any of the systems discussed in Chapter 2.

An additional improvement to Viola's decision-making process is provided by the historical evaluation operators (HEO's). A constraint violation history records all constraint violations handled by Viola, their severity ratings and the actions taken by Viola to resolve them. This is a different approach to the use of history than that

taken by Gray [8] in Diadem. Rather than recording the steps taken to produce the design as in Diadem, Viola's history maintains a record of all previous problem-solving activity. This allows Viola to monitor and evaluate its own problem-solving strategy over the course of the design. This monitoring and evaluation is the role of the HEO's -- by examining the constraint violation history and providing input to Viola's decision-making process, the HEO's help Viola to improve its decisions by avoiding past mistakes and repeating actions that worked in the past. In this way, Viola can also avoid unnecessarily undoing previous corrective actions. Combined with the SAO's, the HEO's allow Viola to make an informed decision that is sensitive to the current problem context.

Objective (c) states that Viola's violation handling strategy should be flexible and explicit. The reason for this objective is that a flexible and explicit strategy is more readily understood and modified by others. Viola's strategy is implemented by the SAO's, the HEO's and the meta-operators. The flexibility of the strategy stems from the fact that the operators are independent and can be readily added or removed to tailor the strategy. Whenever the plan proposal stage is executed, all active operators examine the problem independently and cast their votes if they are relevant. The only interaction between operators comes after the votes have been cast. This interaction is controlled by the meta-operators; therefore, to understand how a new SAO or HEO will interact with potentially many existing SAO's and HEO's, it is only necessary to be familiar with the relatively few meta-operators. By imposing this higher-level control on the HEO's and SAO's, the meta-operators help to make Viola's strategy more explicit.

In combination with a flexible and explicit strategy, a clear separation of domain-dependent and domain-independent knowledge, as stated in objective (d), makes it easier to switch from one domain to another. Viola's focus of attention and plan

proposal stages are domain-independent. The severity ratings, bypass operators, precedence operators, SAO's, HEO's and meta-operators are all applicable to any domain where competing constraints exist and a choice must be made between action (i.e. redesign) and inaction (i.e. relaxation). None of the knowledge in the aforementioned system components makes reference specifically to area or delay constraints or to IC design -- the knowledge refers only to constraints, competing constraints, the importance of those constraints and their violation margins. For example, SAO4 (from Table 5.1) states:

> If the violated constraint has a competitor and the competitor's ranking is greater than the violated constraint's ranking, relaxation should be favoured.

This piece of knowledge is applicable to any situation where a constraint violation is being resolved in the presence of a competing constraint. The only domain-dependent knowledge used by Viola is the knowledge about which constraint types compete and the knowledge used in the plan refinement and implementation stage. The knowledge about which constraint types compete is represented in the *identifyCompetingConstraints* method of each constraint type and the cell redesign knowledge of the plan refinement and implementation stage is represented procedurally. This clear separation of knowledge types should allow transfer of the focus of attention and plan proposal stages to a new domain by defining new competing constraint relationships and adding a new plan refinement and implementation stage. If the new domain must choose between courses of action other than constraint relaxation and redesign, modification of some of the SAO's and HEO's may be required. An example of an alternate domain is project scheduling. In this domain, manpower and time constraints compete and constraint violations are resolved by either constraint relaxation or rescheduling. For project scheduling, the focus of attention stage and the plan proposal stage should be transferrable with little modification because rescheduling can be considered redesign

of the project schedule.

The problems that Viola was tested on were quite small. Objective (e) states that Viola should be computationally efficient to be useful for large designs. The primary means of achieving this efficiency in Viola is the focus of attention stage -- by narrowing Viola's focus to a single constraint violation at a time, considerable computing time is saved from an approach that attempts to resolve all constraint violations in parallel. A similar means of increasing efficiency is the approximation of tracking effects only on the highest ranked competitor in the plan refinement process. This increased efficiency should allow Viola's approach to be used for larger designs; however, Smalltalk, being an interpreted envirionment, runs too slowly to be useful for larger designs. This does not contradict the objective of computational efficiency. The intent was not to develop a system that would be efficient in Smalltalk; rather, the intent was to develop an approach to constraint violation handling that would produce good results with a minimum of computation and would be efficient once translated to a faster programming environment. Smalltalk was a very efficient environment for prototype development, but a production quality system should be implemented in a faster object-oriented system such as C++.

The final objective in Viola's design (objective (f)) states that Viola should enable an inexperienced designer to produce better designs in a shorter time and should reduce the time required for an experienced designer to produce designs. Assuming the author's hand solutions to the problems in Chapter 7 to be the solutions of an experienced designer, Viola succeeded in fulfilling this objective. Viola's solutions to the problems (including time for interactive reconnection of subcells) took an average of 10 to 15 minutes. The corresponding hand solutions averaged 30 to 45 minutes. This time saving will increase with more complex designs and less experienced designers.

In summary, Viola is a flexible, efficient system that is capable of reasoning about design tradeoffs while resolving area and delay constraint violations in IC designs. It produces good designs at a considerable time saving and clearly separates domain-dependent and domain-independent knowledge to allow transfer to other domains. Despite these benefits, there are limitations to Viola's approach to violation handling. These limitations are discussed in the next section.

8.2.2. Limitations Of Viola's Approach

The most difficult problem Viola is required to resolve is the situation where two equally important competing constraints are violated. The objective in this type of problem is to balance the adverse effects on the two competing constraints. Viola uses a heuristic approach (the 50% relaxation heuristic) to address this situation. The results could, however, be improved by better tracking of the effects of resolving a constraint violation on any competing constraints. As each subcell replacement is scheduled, the relative severity of the two constraints could be compared. If the competitor becomes more severe than the constraint violation being handled, the redesign process could be stopped and both constraints could be relaxed. This would produce a more optimal balance between the two competing constraints and would resolve them both in a single violation handling pass.

The problem with directly implementing this approach stems from Viola's severity ratings system. In order to declare one constraint violation more severe than another, the violated constraints and their competing constraints must be assigned severity ratings and the precedence operators must be applied. The application of the precedence operators is necessary because a comparison of constraint violation severity must consider both the severity ratings of the violated constraints and the severity rat-

ings of their competitors. Before assigning new severity ratings to the constraints, it is necessary to reconnect each subcell (manually) and reevaluate all constraints. This combination of factors makes implementation of the revised resolution mechanism within the existing framework prohibitively time consuming. If, however, an automatic reconnection facility were added to STEM and the procedures for reevaluating the constraint network were made more efficient, the added computing time required to incrementally compare the two competing constraints may be justified because both constraint violations could be resolved at the same time.

The quality of Viola's solutions may also be reduced in some cases by the way subcells are chosen for replacement in the plan refinement and implementation stage. Viola currently takes a hill climbing approach to subcell replacement scheduling -- each subcell replacement is chosen based on a local comparison without any overall evaluation of the group of subcell replacements selected. Consider the example in Figure 8.1. An improvement of 10% is required in a violated constraint and three subcell replacements are possible -- each with a different potential improvement in the violated constraint and adverse effect on a competing constraint. With Viola's current hill climbing approach, subcell A would be scheduled for replacement first followed by subcell B. The result would be an improvement of 13% and an adverse effect on the competing constraint of 6%. An alternate best-first search might choose to replace subcells B and C rather than A and B resulting in an improvement of 10% and an adverse effect of 4%. In order for this alternate approach to be used without paying a large penalty in computing time, an improvement to the subcell replacement selection heuristic would also be necessary. The use of net benefit (improvement - adverse effect) as a heuristic would have to be modified to include a penalty for excessive improvement in the violated constraint.

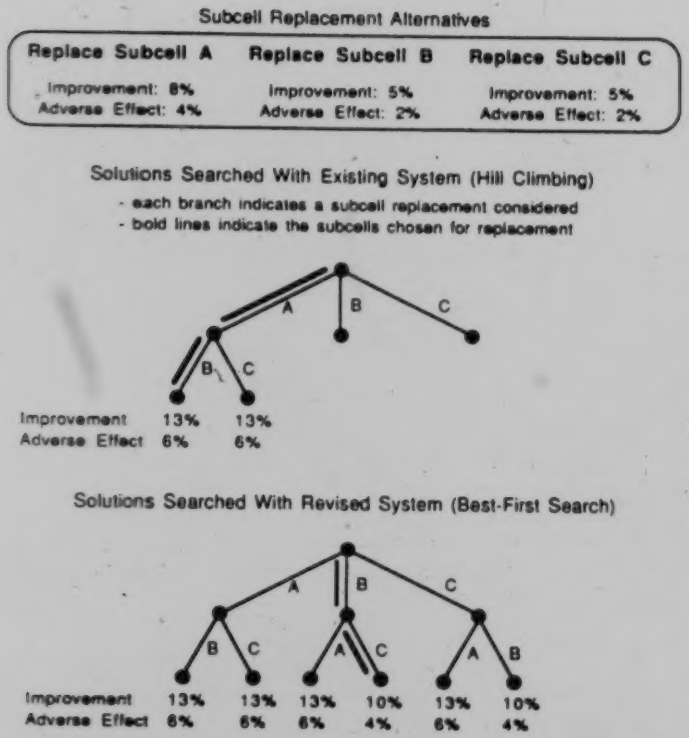


Figure 8.1. An Alternate Subcell Replacement Scheduling Approach

Viola also runs into a hill climbing problem in the focus of attention stage -- the serial methodology of selecting a single constraint violation and resolving it before attempting to resolve other constraint violations is the source of that problem. Because Viola cannot predict the exact effect of resolving a constraint violation on other constraints, it doesn't always make the best possible choice of which constraint violation to work on. In fact, it sometimes creates bigger problems than the one it is trying to solve. Since Viola has no backtracking capability, an additional redesign step is sometimes required to resolve the problem created by the first redesign step. The ability to retract the previous violation handling step could be added to Viola at the expense of

storage overhead. Viola could retain a copy of the original design until a comparison could be made to ensure that the state of the design was improved by Viola's actions.

An extension to this idea of evaluating Viola's decisions before committing to them might be to generate a plan for resolving each constraint violation on the violation list and compare the state of the design resulting from each plan. Viola could then simply implement the best plan. To fully determine the effect of handling each constraint violation on the network, however, would require execution of the plan proposal stage and the plan refinement and implementation stage for each constraint violation on the violation list. In Viola's existing framework, it would be necessary to record the original design as well as the design resulting from resolving each constraint violation. This would introduce a large storage overhead to the system; however, modifying the system to record only the design changes actually made in each case rather than the entire design would alleviate the storage problem. The biggest problem to be overcome in this alternate approach would be the extra computation required to generate each solution for comparison. It seems that the best solution to this problem may be to avoid the backtracking problem by improving Viola's predictive capabilities. In determining the most severe constraint violation, Viola looks at the importance and violation margin of competing constraints as a means of predicting the potential adverse effects of handling a constraint violation. It may be necessary also to consider the subcell replacements available when selecting the most severe constraint violation.

As described in the previous section, the combination of SAO's, HEO's and meta-operators constitute a flexible and explicit violation handling strategy. There is, however, some potential for knowledge to be lost or hidden through the use of operators. As with other rule-based knowledge representations, the knowledge represented by the operators themselves is explicit, but the justification behind that knowledge may not be apparent. This problem could be addressed by good documentation practices --

a justification field could be added to each operator in which the person adding the operator would record a statement describing the reason behind the operator. It would then be up to the person administering the system to ensure that anyone adding an operator records the justification behind that operator.

Most of the limitations discussed in this section stem from the shortcuts taken by Viola for the sake of computational efficiency. As a result, resolution of the limitations will likely have an adverse effect on efficiency. Any future research intended to improve upon the work described in this thesis should look at more computationally efficient means of adressing these limitations. Additional suggestions for future research are presented in the following section.

8.3. Suggestions For Future Research

In addition to making the improvements discussed in the last section, there are a number of other possible directions for future research to take. Some research topics involve enhancements to Viola's capabilities to make it applicable to a broader range of problems and some involve incorporation of Viola with other design automation systems.

In light of the fact that Viola was developed with the intent of being transferrable to other violation handling domains, a logical extension of this research would be to attempt such a transfer. Since the knowledge used in the focus of attention stage (severity rating sytem, bypass operators and precedence operators) and the plan proposal stage (severity assessment operators, historical evaluation operators and meta-operators) is domain-independent, these two stages, which constitute Viola's decision-making process, could be transferred directly to the new domain. The procedural knowledge of the plan refinement and implementation stage would have to be replaced

with knowledge appropriate to the new domain. An appropriate domain for application of Viola's methodology is any domain where competing constraints exist and a decision must be made between taking corrective action (i.e. redesign) and ignoring the problem (i.e. relaxation) to resolve constraint violations. Project scheduling is a suitable alternate domain with considerable profit potential -- many corporations spend a considerable amount of time and money on project scheduling. In the project scheduling domain, a decision must be made between rescheduling and constraint relaxation to resolve competing due date, manpower, cost and equipment availability constraints. Other domains include large design or synthesis tasks such as the design of telephone switching networks. This task involves selection of equipment from one or more manufacturers to meet competing cost and performance constraints. As with IC design, a decision must be made between relaxation and redesign to resolve constraint violations.

Extension to other domains will require the addition of new constraint types to Viola. Viola represents constraints as objects capable of sending and receiving constraint propagation messages, testing themselves for satisfaction, and identifying competing constraints. In most cases, new constraint types will inherit the ability to send and receive propagation messages and test for satisfaction; however, the competing constraint types will have to be specified for the new constraints. In some domains, competition between constraint types will be easy to identify as with area and delay in IC design; however, not all competition will be this simple. To handle more complex competition situations, a more sophisticated mechanism for representing competition relationships will be required. An example of a more complex competition relationship is the case where a system is being designed from a number of available pieces of equipment with constraints on equipment size, equipment performance and equipment cost. In the absence of performance constraints, size and cost may not compete, and

in the absence of size constraints, performance and cost may not compete. A combination of a size constraint and a performance constraint will, however, compete with any cost constraint. Perhaps, each competition relationship will have to be represented explicitly in this way.

In transferring Viola to any large, real-world domains, it will be also be necessary to translate it into a more efficient programming language such as C++.

The first logical combination of Viola with another design automation system would be to add an automatic reconnection facility to STEM. This would speed things up greatly -- the user would no longer have to pause and manually reconnect any replaced subcells after each redesign step taken by Viola.

An interesting research topic would be to use Viola in conjunction with a logic synthesis system. The synthesis system could generate an IC cell from functional specification without worrying about area and delay constraints. Viola could then go through and redesign the generated cell as required to meet the design constraints. The incorporation of a logic minimization system between the synthesis system and Viola would also help to produce a better final circuit.

Viola is a good prototype system for constraint violation handling. Incorporation of any or all of the improvements mentioned should make Viola a useful system for IC design and for other domains.

References

[1] Borning, A. (1981), The Programming Language Aspects Of ThingLab, A Constraint-Oriented Simulation Laboratory, *ACM Transactions On Programming Languages And Systems*, Vol. 3, No. 4, 353-387.

[2] Brown, D.C. & Chandrasekaran, B. (1986), Knowledge And Control For A Mechanical Design Expert System, *Computer*, Vol. 19, No. 7, 92-100.

[3] Daga, A. & Birmingham, W.P. (1990), Failure Recovery In The MICON System, *Proceedings Of The 27th Design Automation Conference*.

[4] Fox, M.S. (1986), Observations On The Role Of Constraints In Problem Solving, *Proceedings Of The Sixth Canadian Conference On Artificial Intelligence*, 172-187.

[5] Girczyc, E.F. & Ly, T.A. (1987), STEM: An IC Design Environment Based On The Smalltalk Model-View-Controller Construct, *Proceedings Of The 24th Design Automation Conference*.

[6] Goldberg, A. & Robson, D. (1989), *Smalltalk-80: The Language*, Addison-Wesley, Don Mills, Ontario.

[7] Goldberg, A. & Robson, D. (1984), *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, Don Mills, Ontario.

[8] Gray, M.A. (1987), Implementing An Intelligent Design Machine In A TMS-Based Inferencing System, *Proceedings Of The 1987 IEEE International Conference On Systems, Man And Cybernetics*, 163-172.

[9] Kelly, V.E. (1984), The Critter System: Automated Critiquing Of Digital Circuit Design, *Proceedings Of The 21st Design Automation Conference*, 419-425.

[10] Ly, T.A. (1989), Managing Design Interactions With Constraint Propagation In An Object-Oriented IC Design Environment, M.Sc. Thesis, The University Of Alberta.

[11] Mitchell, T.M., Steinberg, L.I. & Shulman, J.S. (1985), A Knowledge-Based Approach To Design, *IEEE Transactions On Pattern Analysis And Machine Intelligence*, Vol. PAMI-7, No. 5, 502-510.

[12] Mittal, S., Dym, C.L & Morjara, M. (1986), PRIDE: An Expert System For The Design Of Paper Handling Systems, *Computer*, Vol. 19, No. 7, 102-114.

[13] Murty, S.S. & Addanaki, S. (1987), PROMPT: An Innovative Design Tool, *AAAI-87* 637-642.

[14] Shibahara, T. (1985), On Using Causal Knowledge To Recognize Vital Signals: Knowledge-Based Interpretation Of Arrhythmias, *Proceedings Of The Ninth International Joint Conference On Artificial Intelligence*, 307-314.

[15] Simoudis, E. (1989), A Knowledge-Based System For The Evaluation And Redesign Of Digital Circuit Networks, *IEEE Transactions On Computer-Aided Design*, Vol. 8, No. 3, 302-315.

[16] Steinberg, L.I. (1987), Design = Top Down Refinement Plus Constraint Propagation Plus What?, *Proceedings Of The 1987 IEEE International Conference On Systems, Man And Cybernetics*, 498-502.

[17] Stefik, M. (1981), Planning With Constraints (MOLGEN: Part 1), *Artificial Intelligence*, 16, 111-140.

[18] Stefik, M. (1981), Planning And Meta-Planning (MOLGEN: Part 2), *Artificial Intelligence*, 16, 141-170.

[19] Sussman, G.J. & Steele, G.L. Jr. (1980), CONSTRAINTS - A Language For Expressing Almost Hierarchical Descriptions, *Artificial Intelligence*, 14, 1-39.

[20] Vladimirescu, A., Cohen, E. & Pederson, D.O. (1974), SPICE2 User's Guide, Electronics Research Laboratory, University Of California, Berkely.

Appendix I - Test Case Transcripts

Record Of Violation Handling For Test Case 1

Constraints:

TestCellA.area
- actual value: 40580.3
- constrained value: 38000.0
- importance: Very Important

TestCellA.delayFrominatoout
- actual value: 8.51e-8
- constrained value: 9.0e-8
- importance: Important

TestCellA.delayFrominbtoout
- actual value: 6.76e-8
- constrained value: 7.0e-8
- importance: Important

TestCellA.delayFrominctoout
- actual value: 7.81e-8
- constrained value: 8.0e-8
- importance: Important

Actions Taken By Viola:

Handling constraint: a MaximumAreaPredicate (TestCellA.area)
- violation margin: 6.79023
- redesign list before MOs: OrderedCollection (SAO1 SAO3)
- relaxation list before MOs: OrderedCollection (SAO5)
- redesign list after MOs: OrderedCollection (SAO1 SAO3)
- relaxation list after MOs: OrderedCollection ()
- decided to redesign
- replaced subcells: Nand2B.4 (a Nand2B) Nand3B.3 (a Nand3B) Nand2B.2 (a Nand2B) Nand2B.5 (a Nand2B)

Handling constraint: a MaximumDelayPredicate (TestCellA.delayFrominctoout)
- violation margin: 11.125
- redesign list before MOs: OrderedCollection (HEO5 SAO1 SAO3)
- relaxation list before MOs: OrderedCollection (HEO3)
- redesign list after MOs: OrderedCollection ()
- relaxation list after MOs: OrderedCollection (HEO3)
- decided to relax

Handling constraint: a MaximumDelayPredicate (TestCellA.delayFrominbtoout)
- violation margin: 10.5714
- redesign list before MOs: OrderedCollection (HEO6 HEO5 SAO1 SAO3)
- relaxation list before MOs: OrderedCollection (HEO3)

- redesign list after MOs: OrderedCollection ()
- relaxation list after MOs: OrderedCollection (HEO3)
- decided to relax

Handling constraint: a MaximumDelayPredicate (TestCellA.delayFrominatoout)
- violation margin: 1.22222
- redesign list before MOs: OrderedCollection (HEO6 HEO5 SAO1 SAO3)
- relaxation list before MOs: OrderedCollection (HEO3)
- redesign list after MOs: OrderedCollection ()
- relaxation list after MOs: OrderedCollection (HEO3)
- decided to relax

VIOLATION HANDLING COMPLETE!

Results:

TestCellA.area
- actual value: 37660.0
- constrained value: 38000.0
- importance: Very Important

TestCellA.delayFrominatoout
- actual value: 9.11e-8
- constrained value: 9.11e-8
- importance: Important

TestCellA.delayFrominbtoout
- actual value: 7.74e-8
- constrained value: 7.74e-8
- importance: Important

TestCellA.delayFrominctoout
- actual value: 8.89e-8
- constrained value: 8.89e-8
- importance: Important

Record Of Violation Handling For Test Case 2

Constraints:

TestCellA.delayFrominatoout
- actual value: 8.51e-8
- constrained value: 5.0e-8
- importance: Very Important

TestCellA.area
- actual value: 40580.3
- constrained value: 40000.0
- importance: Important

TestCellA.delayFrominbtoout
- actual value: 6.76e-8
- constrained value: 5.0e-8
- importance: Important

TestCellA.delayFrominctoout
- actual value: 7.81e-8
- constrained value: 5.0e-8
- importance: Not Important

Actions Taken By Viola:

Handling constraint: a MaximumDelayPredicate (TestCellA.delayFrominatoout)
- violation margin: 70.2
- redesign list before MOs: OrderedCollection (SAO1 SAO3)
- relaxation list before MOs: OrderedCollection ()
- redesign list after MOs: OrderedCollection (SAO1 SAO3)
- relaxation list after MOs: OrderedCollection ()
- decided to redesign
- replaced subcells: Nand3B.1 (a Nand3B) InverterB.1 (an InverterB) InverterB.2 (an InverterB) Nand2B.1 (a Nand2B) Nand3B.2 (a Nand3B) Nand3B.3 (a Nand3B) Nand2B.3 (a Nand2B)
- also had to relax by 35.4

Handling constraint: a MaximumAreaPredicate (TestCellA.area)
- violation margin: 30.6539
- redesign list before MOs: OrderedCollection (HEO5 SAO1 SAO3)
- relaxation list before MOs: OrderedCollection (HEO3 SAO5)
- redesign list after MOs: OrderedCollection ()
- relaxation list after MOs: OrderedCollection (HEO3)
- decided to relax

Handling constraint: a MaximumDelayPredicate (TestCellA.delayFrominbtoout)
- violation margin: 9.2
- redesign list before MOs: OrderedCollection (SAO1 SAO3)
- relaxation list before MOs: OrderedCollection (HEO7 HEO4)
- redesign list after MOs: OrderedCollection ()

- relaxation list after MOs: OrderedCollection (HEO7 HEO4)
- decided to relax

Handling constraint: a MaximumDelayPredicate (TestCellA.delayFrominctoout)
- violation margin: 34.2
- redesign list before MOs: OrderedCollection (HEO6 SAO3)
- relaxation list before MOs: OrderedCollection (HEO4 SAO2)
- redesign list after MOs: OrderedCollection (HEO6)
- relaxation list after MOs: OrderedCollection (HEO4)
- decided to relax

VIOLATION HANDLING COMPLETE!

Results:

TestCellA.delayFrominatoout
- actual value: 6.77e-8
- constrained value: 6.77e-8
- importance: Very Important

TestCellA.area
- actual value: 52261.6
- constrained value: 52261.6
- importance: Important

TestCellA.delayFrominbtoout
- actual value: 5.46e-8
- constrained value: 5.46e-8
- importance: Important

TestCellA.delayFrominctoout
- actual value: 6.71e-8
- constrained value: 6.71e-8
- importance: Not Important

Record Of Violation Handling For Test Case 3

Constraints:

TestCellA.area
- actual value: 40580.3
- constrained value: 43000.0
- importance: Very Important

TestCellA.delayFrominatoout
- actual value: 8.51e-8
- constrained value: 8.0e-8
- importance: Important

TestCellA.delayFrominbtoout
- actual value: 6.76e-8
- constrained value: 6.0e-8
- importance: Important

TestCellA.delayFrominctoout
- actual value: 7.81e-8
- constrained value: 7.5e-8
- importance: Important

Actions Taken By Viola:

Handling constraint: a MaximumDelayPredicate (TestCellA.delayFrominbtoout)
- violation margin: 12.6667
- redesign list before MOs: OrderedCollection (SAO1 SAO3)
- relaxation list before MOs: OrderedCollection ()
- redesign list after MOs: OrderedCollection (SAO1 SAO3)
- relaxation list after MOs: OrderedCollection ()
- decided to redesign
- replaced subcells: Nand3B.1 (a Nand3B) InverterB.1 (an InverterB) Nand3B.2 (a Nand3B)

Handling constraint: a MaximumAreaPredicate (TestCellA.area)
- violation margin: 7.53118
- redesign list before MOs: OrderedCollection (HEO12 HEO5 SAO1 SAO3)
- relaxation list before MOs: OrderedCollection (SAO5)
- redesign list after MOs: OrderedCollection (HEO12 HEO5 SAO1 SAO3)
- relaxation list after MOs: OrderedCollection ()
- decided to redesign
- replaced subcells: Nand2B.4 (a Nand2B) Nand2B.3 (a Nand2B) Nand2B.1 (a Nand2B) InverterD.1 (an InverterD)

Handling constraint: a MaximumDelayPredicate (TestCellA.delayFrominatoout)
- violation margin: 8.75
- redesign list before MOs: OrderedCollection (SAO1 SAO3)
- relaxation list before MOs: OrderedCollection (HEO3 HEO7)
- redesign list after MOs: OrderedCollection ()

- relaxation list after MOs: OrderedCollection (HEO3 HEO7)
- decided to relax

Handling constraint: a MaximumDelayPredicate (TestCellA.delayFrominbtoout)
- violation margin: 7.33333
- redesign list before MOs: OrderedCollection (HEO6 HEO5 SAO1 SAO3)
- relaxation list before MOs: OrderedCollection (HEO3 HEO13)
- redesign list after MOs: OrderedCollection ()
- relaxation list after MOs: OrderedCollection (HEO3 HEO13)
- decided to relax

VIOLATION HANDLING COMPLETE!

Results:

TestCellA.area
- actual value: 42770.5
- constrained value: 43000.0
- importance: Very Important

TestCellA.delayFrominatoout
- actual value: 8.7e-8
- constrained value: 8.7e-8
- importance: Important

TestCellA.delayFrominbtoout
- actual value: 6.44e-8
- constrained value: 6.44e-8
- importance: Important

TestCellA.delayFrominctoout
- actual value: 7.4e-8
- constrained value: 7.5e-8
- importance: Important

Record Of Violation Handling For Test Case 4

Constraints:

TestCellA.area
- actual value: 40580.3
- constrained value: 50000.0
- importance: Important

TestCellA.delayFrominatoout
- actual value: 8.51e-8
- constrained value: 6.0e-8
- importance: Important

Actions Taken By Viola:

Handling constraint: a MaximumDelayPredicate (TestCellA.delayFrominatoout)
- violation margin: 41.8333
- redesign list before MOs: OrderedCollection (SAO1 SAO3)
- relaxation list before MOs: OrderedCollection ()
- redesign list after MOs: OrderedCollection (SAO1 SAO3)
- relaxation list after MOs: OrderedCollection ()
- decided to redesign
- replaced subcells: Nand3B.1 (a Nand3B) InverterB.1 (an InverterB) InverterB.2 (an InverterB) Nand2B.1 (a Nand2B) Nand3B.2 (a Nand3B) Nand3B.3 (a Nand3B) Nand2B.3 (a Nand2B)
- also had to relax by 12.8333

Handling constraint: a MaximumAreaPredicate (TestCellA.area)
- violation margin: 4.52313
- redesign list before MOs: OrderedCollection (HEO5 SAO1 SAO3)
- relaxation list before MOs: OrderedCollection (HEO3)
- redesign list after MOs: OrderedCollection ()
- relaxation list after MOs: OrderedCollection (HEO3)
- decided to relax

VIOLATION HANDLING COMPLETE!

Results:

TestCellA.area
- actual value: 52261.6
- constrained value: 52261.6
- importance: Important

TestCellA.delayFrominatoout
- actual value: 6.77e-8
- constrained value: 6.77e-8
- importance: Important

Record Of Violation Handling For Test Case 5

Constraints:

TestCellA.area
- actual value: 40580.3
- constrained value: 39000.0
- importance: Important

TestCellA.delayFrominatoout
- actual value: 8.51e-8
- constrained value: 9.0e-8
- importance: Important

Actions Taken By Viola:

Handling constraint: a MaximumAreaPredicate (TestCellA.area)
- violation margin: 4.05202
- redesign list before MOs: OrderedCollection (SAO1 SAO3)
- relaxation list before MOs: OrderedCollection ()
- redesign list after MOs: OrderedCollection (SAO1 SAO3)
- relaxation list after MOs: OrderedCollection ()
- decided to redesign
- replaced subcells: InverterB.3 (an InverterB) Nand2B.2 (a Nand2B) Nand2B.5 (a Nand2B) Nand2B.4 (a Nand2B)

VIOLATION HANDLING COMPLETE!

Results:

TestCellA.area
- actual value: 38572.6
- constrained value: 39000.0
- importance: Important

TestCellA.delayFrominatoout
- actual value: 8.51e-8
- constrained value: 9.0e-8
- importance: Important

Record Of Violation Handling For Test Case 6

Constraints:

TestCellA.area
- actual value: 40580.3
- constrained value: 40000.0
- importance: Important

TestCellA.delayFrominatoout
- actual value: 8.51e-8
- constrained value: 8.0e-8
- importance: Important

Actions Taken By Viola:

Handling constraint: a MaximumDelayPredicate (TestCellA.delayFrominatoout)
- violation margin: 6.37501
- redesign list before MOs: OrderedCollection (SAO1)
- relaxation list before MOs: OrderedCollection ()
- redesign list after MOs: OrderedCollection (SAO1)
- relaxation list after MOs: OrderedCollection ()
- decided to redesign
- performed 50% relaxation
- replaced subcells: Nand2B.1 (a Nand2B)

Handling constraint: a MaximumAreaPredicate (TestCellA.area)
- violation margin: 4.18852
- redesign list before MOs: OrderedCollection (HEO5 SAO1 SAO3)
- relaxation list before MOs: OrderedCollection ()
- redesign list after MOs: OrderedCollection (HEO5 SAO1 SAO3)
- relaxation list after MOs: OrderedCollection ()
- decided to redesign
- replaced subcells: InverterB.3 (an InverterB) Nand2B.2 (a Nand2B) Nand2B.5 (a Nand2B) Nand2B.4 (a Nand2B)

VIOLATION HANDLING COMPLETE!

Results:

TestCellA.area
- actual value: 39667.7
- constrained value: 40000.0
- importance: Important

TestCellA.delayFrominatoout
- actual value: 8.21e-8
- constrained value: 8.255e-8
- importance: Important

Record Of Violation Handling For Test Case 7

Constraints:

TestCellA.area
- actual value: 40580.3
- constrained value: 40000.0
- importance: Important

TestCellA.delayFrominatoout
- actual value: 8.51e-8
- constrained value: 7.0e-8
- importance: Important

TestCellA.delayFrominbtoout
- actual value: 6.76e-8
- constrained value: 6.5e-8
- importance: Important

TestCellA.delayFrominctoout
- actual value: 7.81e-8
- constrained value: 7.0e-8
- importance: Important

Actions Taken By Viola:

Handling constraint: a MaximumDelayPredicate (TestCellA.delayFrominatoout)
- violation margin: 21.5714
- redesign list before MOs: OrderedCollection (SAO1)
- relaxation list before MOs: OrderedCollection ()
- redesign list after MOs: OrderedCollection (SAO1)
- relaxation list after MOs: OrderedCollection ()
- decided to redesign
- performed 50% relaxation
- replaced subcells: Nand3B.1 (a Nand3B) InverterB.2 (an InverterB) InverterB.1 (an InverterB)

Handling constraint: a MaximumDelayPredicate (TestCellA.delayFrominctoout)
- violation margin: 1.7143
- redesign list before MOs: OrderedCollection (SAO1)
- relaxation list before MOs: OrderedCollection (HEO7 HEO4)
- redesign list after MOs: OrderedCollection ()
- relaxation list after MOs: OrderedCollection (HEO7 HEO4)
- decided to relax

Handling constraint: a MaximumAreaPredicate (TestCellA.area)
- violation margin: 16.0523
- redesign list before MOs: OrderedCollection (SAO1 SAO3)
- relaxation list before MOs: OrderedCollection (SAO5)
- redesign list after MOs: OrderedCollection (SAO1 SAO3)
- relaxation list after MOs: OrderedCollection ()

- decided to redesign
- used 50RH to relax by 6.80936
- replaced subcells: Nand2B.2 (a Nand2B) Nand2B.4 (a Nand2B) Nand2B.1 (a Nand2B) Nand2B.3 (a Nand2B) InverterD.2 (an InverterD)

Handling constraint: a MaximumDelayPredicate (TestCellA.delayFrominatoout)
- violation margin: 8.57511
- redesign list before MOs: OrderedCollection (HEO6 HEO5 SAO1 SAO3)
- relaxation list before MOs: OrderedCollection ()
- redesign list after MOs: OrderedCollection (HEO6 HEO5 SAO1 SAO3)
- relaxation list after MOs: OrderedCollection ()
- decided to redesign
- performed 50% relaxation
- replaced subcells: InverterB.1 (an InverterB) Nand2A.3 (a Nand2A)

Handling constraint: a MaximumDelayPredicate (TestCellA.delayFrominbtoout)
- violation margin: 1.07693
- redesign list before MOs: OrderedCollection (SAO1)
- relaxation list before MOs: OrderedCollection (HEO7)
- redesign list after MOs: OrderedCollection ()
- relaxation list after MOs: OrderedCollection (HEO7)
- decided to relax

Handling constraint: a MaximumAreaPredicate (TestCellA.area)
- violation margin: 5.23602
- redesign list before MOs: OrderedCollection (SAO1 SAO3)
- relaxation list before MOs: OrderedCollection (HEO7 HEO4 SAO5)
- redesign list after MOs: OrderedCollection ()
- relaxation list after MOs: OrderedCollection (HEO7 HEO4)
- decided to relax

VIOLATION HANDLING COMPLETE!

Results:

TestCellA.area
- actual value: 44960.8
- constrained value: 44960.8
- importance: Important

TestCellA.delayFrominatoout
- actual value: 7.84e-8
- constrained value: 8.0875e-8
- importance: Important

TestCellA.delayFrominbtoout
- actual value: 6.57e-8
- constrained value: 6.57e-8
- importance: Important

TestCellA.delayFrominctoout

- actual value: 7.12e-8
- constrained value: 7.12e-8
- importance: Important